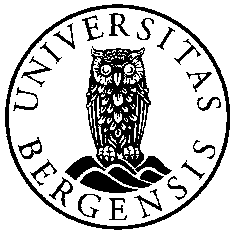
En visuell tilnærming til simulering av købaserte modeller i Java

Morten Johannes Ervik
Institutt for Informatikk
Universitetet i Bergen
5020 BERGEN



Oppgave for graden master i informatikk

14. juni 2004

Sammendrag

I denne oppgaven ser vi på diskret hendelsessimulering (DHS) (*Discrete event simulation*) og Java. Vi vil vise at Java allerede har en del egenskaper og redskaper som for å jobbe med DHS. Dette gjør at vil vi implementere et rammeverk (mjmod - *my java modelling*) for diskret hendelsessimulering i dette språket. Mønsteret vi har valgt for gruppering av hendelser for dette er en modifisert utgave av den såkalte tre-fase-tilnærmingen til aktivitetsinterrogering (*Activity Scanning*). Dette er mindre utbredt enn hendelsesplanlegging (*Event scheduling*) og prosessinteraksjon (*Process Interaction*), men vi vil vise at dette egner seg godt til en visuell tilnærming til modelleringen.

For å spesifisere modellene som skal simuleres vil vi utvikle et XML-basert språk (mjml - *my java modelling language*). Modellene kan man dermed sette opp i generelle tekst- eller XML-redigeringsprogram. Vi vil dessuten utvikle et grafiske grensesnitt for å koble dem opp. Dette vil vi tilegne en del funksjonalitet som lar oss studere og manipulere simuleringer under kjøring.

Vi fokuserer på en gruppe av problemer som kan løses med købasert modellering (*Queueing Modelling*). Disse problemene kan reduseres til modeller bygget opp av et nettverk av køer og aktiviteter. For å visualisere logiske strukturer i modeller laget for mjmod vil vi implementere en oversetter fra mjml til et generelt filformat for å spesifisere grafer kalt GraphXML.

Vi vil ta for oss noen utvalgte simuleringsproblem for å demonstrere mjmod i bruk.

Forord

Dette er en oppgave til graden Master of Science innen fagfeltet optimering ved Institutt for Informatikk, Universitetet i Bergen.

Jeg vil takke alle som har hjulpet meg under utformingen av denne oppgaven. Spesielt takk til min veileder Trond Steihaug.

Takk til Johannes Hoff og Erik Faugstad for å lese gjennom oppgaven og rette skrivefeil og komme med forslag til andre utbedringer.

Takk til Visekongene og ModPlay@ChrMich for å ha redusert de kreative kravene i arbeidsomme stunder, men også for kjærkomne avbrekk fra diskrete hendelser.

Takk til mine foreldre for støtte og for alltid å stille opp når det trengs.

Takk også til Juliette Flore Renaud for å ha tålmodighet med meg under arbeidet med denne oppgaven, samt forståelse for at kreativiteten på kjøkkenet har vært laber en stund. (Og ikke minst for å gi meg en powerball, som nok har hindret meg i å utvikle den visse senebetennelse.) Merci!

Morten Johannes Ervik,

Bergen, 13. juni 2004

Innhold

Figurer

Kapittel 1

Innledning

1.1 Bakgrunn

1.1.1 Simulering

Med simulering menes gjerne - løst forklart - å "etterligne et (konstruert) hendelsesforløp". Dette er noe menneskeheten har drevet med siden Newtons tid.[35] Ved å sette opp matematiske modeller prøver man å finne analytiske løsninger som gjør det mulig å forutse oppførselen til et system over tid. Siden de digitale datamaskinenes inntog på førtitallet kan dette gjøres i stadig større skala. Disse maskinene har ingenting imot å bli prakket på uendelige repetetive matematiske oppgaver.[1] Antall transistorer per kvadrattomme på en integrert krets har siden da blitt omlag fordoblet hver attende måned[2], og datamaskinenes regneevne har altså økt i tilsvarende tempo.

I følge bokmålsordboken: "simulere : ˜e´re v2 (fra lat., besl med *simili-*) forstille seg, late som *s- syk* / etterligne et (konstruert) hendelsesforløp (ved hjelp av datamaskin) *det er utviklet en modell for å s- driften av fiskebåter*."

1.1.2 Hvorfor simulere

Grunnene til å gjøre simuleringer i stedet for å utføre virkelige eksperimenter kan være mange. Eksperimentene kan for eksempel være svært kostbare å gjennomføre i den "virkelige verden". Kanskje krever de mye råmaterialer, plass eller tid. Forskning innen fysikk, kjemi og biologi bruker simulering i utstrakt grad.

En annen grunn for simulering er i beslutnigstagningsøyemed. Vi kan benytte oss av simuleringer for å finne sannsynligheten for fremtidige hendelser og handle deretter. Det kan være alt fra værmeldinger for påskefjellet, via sykehusdrift til hvordan bilkøer blir påvirket ved endringer av strukturer i en bilvask. Økonomer bruker simuleringer i sine studier av produksjon, fordeling og forbruk av varer og tjenester.

Etiske grunner kan også ligge bak valget av simuleringer fremfor virkelige eksperimenter, som i Ørjan Bergmann[6] sine eksperimenter på navigeringsevnene til rotter.

[1] Boken *"The Joy of π"* av David Blatner har en passasje om D. F. Fergeuson som brukte et år av livet sitt på å få regnet seg til å inneha verdensekorden i antall desimaler i π. I 1947 la han frem 808 sifre regnet ut på en mekanisk skrivebordskalkulator. I 1949 kom ENIAC opp med 2037 sifre på 70 timer.

[2] Jfr George Moore sin - ofte feilsiterte - lov fra 1965.

 *"...almost every shot involves some kind of simulation program or simulated movement."*
-Oren Jacob, fra Pixar Animation Studios, om *Finding Nemo*

Figur 1.1: Simulert vann i *"Finding Nemo"*

Et av de nyere bruksområdene til simulering er innen underholdningsbransjen. Peter Jackson, for eksempel, benytter datakraft på blant annet simulering for å levendegjøre orkehordene i "Ringenes Herre"-filmene og ofre for kjempeapen King Kong i den kommende filmen om ham.

Et mer klassisk problem for simulering er modellering av bevegelsene til vann, som er imponerende realistisk gjort i verker som den helaftens strålefulgte[3] *Finding Nemo* av Pixar (se figur 1.1), men som har sin noe mer praktiske anvendelse i oseanografi og maritim design.

Dataspill som for eksempel *SimTower* - der spillet går i stor grad ut på å simulere heiser, *SimCity* - der man simulerer byplanlegging og *Pontifex* - der man simulerer brobygging er alle enkle simulatorer pakket inn som spill.

1.1.3 Diskret hendelssessimulering

Vi kan kategorisere modellene som simuleres på flere måter. De kan være stokastiske (innslag av tilfeldigheter gjør at man får forskjellig utfall selv ved identiske initialverdier) eller deterministiske (alltid samme utfall, gitt samme initialverdier), diskrete eller kontinuerlige, de som baserer seg på ulike former for differensialligninger eller ikke.

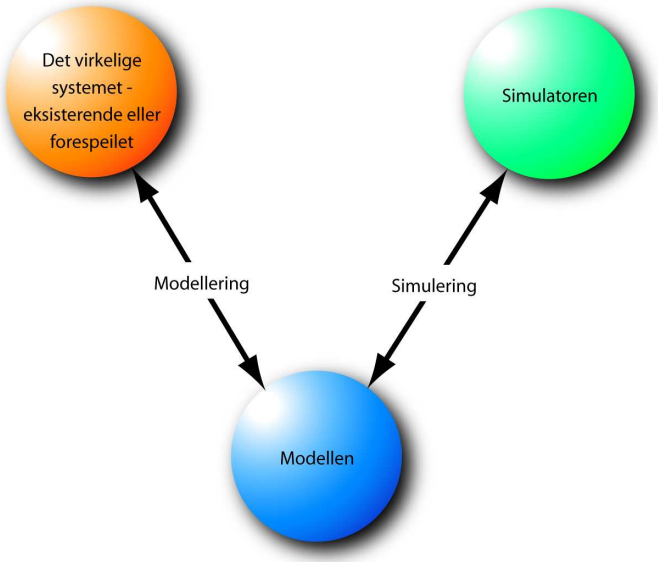
Vi skal fokusere på grenen innen simulering kalt diskret hendelsessimulering (*Discrete event simulation*). Kort forklart er dette et system for simulering der simulasjonstiden springer fra relevante hendelser til relevante hendelser, i motsetning til kontinuerlig simulering der man tenker seg at tiden flyter jevnt og ting hender som funksjon av den.

1.1.4 Det virkelige systemet, modellen og simulatoren

Zeigler[35] tenker seg at modellering og simulering innebefatter tre ting (Se figur 1.2.):

- *Det virkelige systemet* som skal modelleres kan være eksisterene eller bare forespeilet. Dette er i hovedsak en kilde til data.

[3] En kan argumentere for at all *strålefulgt* kunst er simulering - simulering av hvordan lyset brer seg.



Figur 1.2: Sammenhenger i modellering og simulering.

- *Modellen* er instruksjoner som kan generere data sammenlignbare med data fra det virkelige systemet. *Strukturen til modellen* er dette settet av instruksjoner. *Oppførselen* til modellen er alle mulige data som kan bli generert ved å kjøre instruksjonene til modellen.
- *Simulatoren* utfører instruksjonene gitt av modellen og dermed genererer dens oppførsel.

Videre spesifiserer han to sammenhenger mellom disse:

- *Modelleringssammenhengen* er en kobling mellom det virkelige systemet og modellen. Denne forteller i hvilken grad modellen representerer systemet man vil modellere. Generelt kan en modell sies å samsvare med det virkelige systemet dersom data generert gitt instruksjonen til modellen samsvarer med data fra det virkelige systemet.
- *Simulasjonssammenhengen* er en kobling mellom modellen og simulatoren. Den forteller i hvilken grad simulatoren kan utføre instruksjonene til modellen.

1.2 Motivasjon

Siden en modell er en formalisme, er den nødvendigvis en abstraksjon. Paul et al[21] sier:

> "Intet simuleringsprogram kan modellere alle former for systemoppførsler uten å gjøre forenklinger eller modifikasjoner."

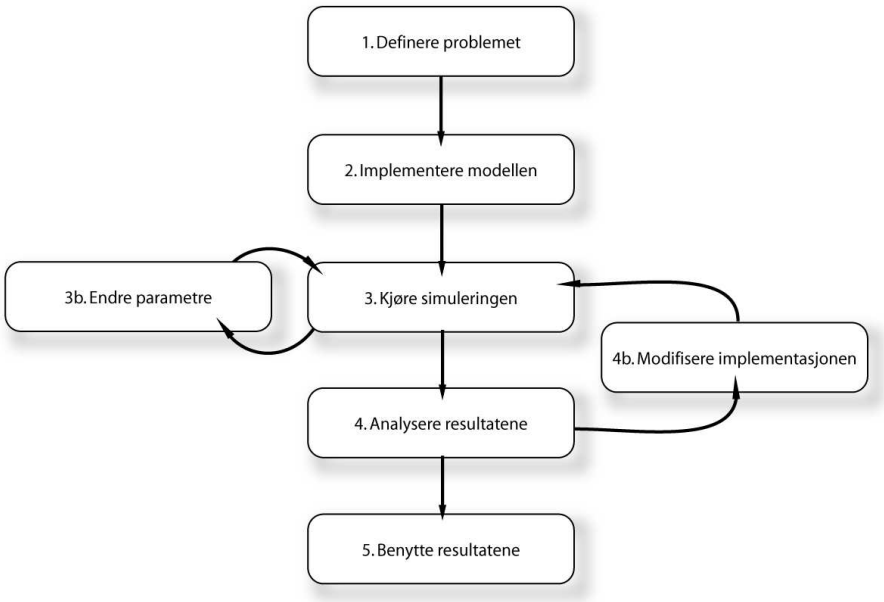
Å velge hvilken måte man skal spesifisere modeller i et modelleringsverktøy, vil altså alltid måtte basere seg på kompromisser. Generalitet må vike for oversiktlighet og enkelhet. Vi kan dermed argumentere for at den beste måten å skape simulasjonssystemer er å kode dem fra bunnen av i et programmeringsspråk, men dette krever førstehånds kunnskap til programmering.

1.2.1 Hvorfor en visuell tilnærming til simulering

Et alternativ kan da være å skape modeller for simulering grafisk, men begrense hvilke former for systemer som kan modelleres. Vi kan tenke oss at vi kan koble opp modellene våre i et grafisk grensesnitt uten å måtte se programmeringsspråk i det hele tatt. Dette har en rekke konsekvenser:

- Det å ha et kart over modellen reint visuelt er en stor fordel når en skal analysere modellens korrekthet. Å feilsøke en simuleringsmodell og finne mulige forbedringer blir dermed lettere.
- Vi kan konsentrere oss om å få logikken korrekt i modellene og sleppe å tenke på de underliggende implementasjonstekniske detaljene. Dette gjør det godt egnet blant annet i læresammenhenger.
- Et grafisk orientert simuleringsprogram kan gjøre det mulig for ikke-spesialister innenfor feltet å sette simuleringer.

For å utdype hvilke fordeler dette kan medføre vil vi vise til gangen i et simuleringsprosjekt.



Figur 1.3: Stadier i et simuleringsprosjekt

Figur 1.3 er en oversikt over hvilke stadier en simulering kan brytes ned til. I første stadiet defineres problemet man vil simulere. I stadie to implementeres dette i valgte simuleringsprogram. Etter å ha kjørt simuleringen i det tredje stadiet vil man gjerne justere betingelsene for modellene for å skaffe flere data til analysen av dem. Etter å ha analysert resultatene i stadie 4 må vi gjerne justere implementasjonen vår, for så å kjøre simuleringen om igjen. Disse løkkene gjentas til man har tilstrekkelig med resultater til å kunne benytte dem til ønsket formål - for eksempel å ta en avgjørelse.

Det vi vil se på i denne oppgaven er altså i hvilken grad grafiske redskaper kan hjelpe oss i de tre første stadiene av simuleringsprosjektet nevnt over; "Definere problemet", "Implementere modellen", og "Kjøre simuleringen".

Vi vil begrense oss til å se på en undergruppe av diskret hendelsessimulering; købasert modellering.

1.2.2 Købasert modellering

Idéen om købasert modellering ble opprinnelig klekket ut for å studere problemer innen telefoni. Dette har vist seg å være et nyttig redskap for å se på en rekke problemstillinger innen diskret hendelsessimulering. Systemer der:

- ruting av *enheter*
- sekvensering av jobber for *tjenere* som gjør arbeid på disse enhetene
- sekvensering av *ressurser* som behøves av disse tjenerene

er det som i hovedsak avgjør *ventetider*, *antall enheter* i systemet, *resursutnyttelse* og *gjennomløp* kan løses av købaserte modeller. (Fishman[9], side 8). Dette kan også kalles *prosessorientert* simulering.

1.2.3 Hvorfor Java

Det å velge et objektorientert programmeringsspråk som Java for implementasjonen av et simulasjonssystem gjør at vi kan benytte oss av objektorienterte programmeringsteknikker som klasser, arv, polimorfisme og innkapsling. Samtidig har Java et stort standardbibliotek som inneholder blant annet dynamiske datastrukturer og redskaper for å sette opp grafiske brukergrensesnitt.

1.3 Mål

Målet med denne oppgaven er å implementere en prototype på et simuleringsprogram i Java som lar oss studere modellene visuelt (mjmod - *my java modelling*) - både den logiske strukturen i dem og hvordan de oppfører seg under simulering.

For å oppnå dette vil vi først se på ulike tilnærminger til diskret hendelsessimulering for å finne en som egner seg godt for å representere modellene visuelt. Vi vil argumentere for å bruke et mønster for gruppering av hendelser basert på aktivitetsinterrogering.

Vi vil se på hvilke egenskaper og redskaper Java allerede har som kjennetegner systemer for å jobbe med diskret hendelsessimulering. Videre vil vi implementere et rammeverk for dette i Java.

Vi vil vise hvorfor XML kan være et godt valg av språk for å spesifisere modellene våre. Ut fra dette vil vi utvikle et XML-basert språk (mjml - *my java modelling language*) for å spesifisere modellene som skal simuleres. Vi vil også lage et eget grensesnitt for å *koble opp* modeller til mjmod. Dette grensesnittet kan brukes til å overvåke ønskede deler av modellene under kjøring av simuleringen.

For å visualisere logiske strukturer i modeller laget for mjmod vil vi implementere en oversetter fra mjml til et generelt filformat for å spesifisere grafer kalt GraphXML. Vi vil også vise en del funksjonalitet ved grenesnittet for å overvåke deler av systemet under kjøring. Vi vil teste ut ideer rundt hvordan basiskomponenter dermed kan bygge opp komplekse købaserte modeller og slik modellere ulike scenarioer.

1.4 Målgruppe

Enhver bachelor innen informatikk eller tilsvarende skal kunne få utbytte av å studere denne oppgaven. Vi forutsetter ingen forkunskaper innen simulering, men en generell forståelse for problemstillinger innen informatikk kan derimot komme godt med.

1.5 Oppbygging

I del I gir vi en kort generell innføring i diskrete hendelsessimulering, for så å vise til ulike tilnærminger til denne grenen innen simulering og hvordan objektorientert programmering kan hjelpe oss. Del II følger så med en utdypning av simulasjonssystemet vi har utviklet, før vi runder av med noen utvalgte eksempler på dette i bruk i del III.

Del I

Hvordan angripe diskret hendelsessimulering i Java

I denne delen tar vi for oss det teoretiske grunnlaget for denne oppgaven. Vi vil gå nærmer inn på hva vi mener med diskret hendelsessimulering, samt definere begrepene som brukes i resten av teksten. Deretter ser vi på de mest utbredte tilnærmingene til objektinteraksjon og hvilken som passer best for vårt formål, før vi går gjennom ulike måter å spesifisere en modell til et simuleringssystem. Vi vil også gi en kjapp oversikt over noen allerede eksisterende simuleringsprogram før delen avsluttes med å vise frem deler av programmeringspråket Java som er spesielt interessante for oppgaven vår.

"Still one wonders, 'How finely will we need to copy the brain to achieve AI?' The Real answer is probably that it all depends on how many features of human consciousness you want to simulate."

Douglas R. Hofstadter i "Gödel, Escher, Bach: an Eternal Golden Braid

Kapittel 2

Diskret hendelsessimulering

I dette kapittelet vil vi utdype begrepet diskret hendelsessimulering og bla. se på ulike tilnærminger. Men først; terminologi.

2.1 Terminologi

Terminologi innen feltet diskret hendelsessimulering er i mindre grad etablert enn for vitenskaper som for eksempel fysikk og matematikk, derfor ser jeg det som fruktbart å definere noen begreper før vi går videre.

2.1.1 Simulasjonsleder

Simulasjonslederen (*Simulation executive*) er kontrollorganet i simulatoren den gir ordet til de prosesser, aktiviteter eller hendelser som står for tur til å utføres, alt etter hvilke verdenssyn man har i simulasjonssystemet.

2.1.2 Attributt

En *attributt* er en verdi, eller på andre måter målbare størrelser, i modellen som simuleres. For eksempel antall enheter i en kø.

2.1.3 Systemets tilstand

Samlingen av alle attributter i en simulering til enhver tid kalles *systemets tilstand.*

2.1.4 Hendelse

Vi har en *hendelse* når systemets tilstand endrer seg. For eksempel det at en enhet ankommer en kø på kan sees på som en hendelse.

2.1.5 Simulasjonstid

Tiden internt i systemet som modelleres kalles *simulasjonstid.* Denne er ikke lineær, men snarere springer den fra hendelse til hendelse.

2.1.6 Hendelseskø

Hendelseskøen er køen som holder rede på fremtidige hendelser i et diskret hendelsessystem. Dette innebærer at den opprettholder en liste med hendelser og returnerer på forespørsel den som er planlagt å starte først.

2.1.7 Brukertid

Tiden som blir oppfattet av brukeren, kaller vi *brukertid*. Altså hvor lang tid datamaskinen bruker på å kjøre systemet vi setter den til å simulere. Dette blir et mål på hvor effektivt programmet/simuleringen er, men vi kommer også til å benytte dette i forbindelse med det grafiske brukergrensesnittet. En annen term man ofte bruker om dette er kjøretid, men det kan bli skape misforståelser når vi snakker om simulering da modellen også *kjøres*.

2.2 Formalismer

Et system for modellering baseres nødvendigvis på en måte å abstrahere tidsflyten på.

2.2.1 Lindleys formel

Simuleringer innen feks. fysikk nytter *calculus* og differensialligninger for å modellere problemer. Dette kan vi også nytte til købaserte diskrete hendelsessimuleringer ved å bruke *Lindleys formel* (Fishman[9], side 28), men det blir fort kompliserte saker da for eksempel antall bivilkår øker eksponentielt med antall *tjenere* som er involvert. Derfor er det vanlig å benytte andre tilnærminger.

2.2.2 DEVS-formalisme

Zeigler[35] innførte i artikkelen *Object-Oriented Modelling and Discrete-Event Simulation* et formelt rammeverk for diskrete hendelsessimuleringer han kalte DEVS (*Discrete Event System Specification*). Dette spesifiserer diskrete hendelsessimuleringer som matematiske objekter, systemer med tidsbase, inndata, tilstander og utdata. Dessuten har systemet metoder/funksjoner for å finne neste tilstand og hva som skal bli utdata gitt tilstanden systemet har nå og dets inndata.

Basismodeller

Man bygger større modeller i DEVS ved å definere basismodeller for så å sette dem sammen i et hierarkisk system. Zeigler definerer basismodellene som følger.

En DEVS er en struktur:

$M =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau a >$.

X er settet av eksterne (inndata) hendelsestyper.

S er det sekvensielle tilstands-settet.

Y er utdata-settet.

$\delta_{int} : S \rightarrow S$,den interne overgangsfunksjonen.

$\delta_{ext} : Q \times S \rightarrow S$, den eksterne overgangsfunksjonen der Q er det totale settet $= \{(s, e) | s \in S, 0 <= e <= \tau a(s)\}$.

$\tau a : S \rightarrow R^{+}_{0,\infty}$, tidsavanseringsfunksjonen.
$\lambda : S \rightarrow Y$, utdata-funksjonen.

Koblede modeller

Siden kan man koble sammen slike basismodeller slik:

$DN =< D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$,der D er et sett av komponentnavn; for hver i i D:

M_i er en basismodell-komponent

I_i er et sett, innflytelsene til i; og for hver j i I_i,

Z_{ij} er en funksjon, i-til-j-utdata-funksjonen og

select er en funksjon, "tie-break"-velgeren som bla. forhindrer *vranglåser*.

2.3 Objektinteraksjon og hendelsesflyt

Selv den enkleste diskret hendelsessimulator innbefatter store mengder hendelser som skal holdes rede på i systemet. Derfor er det kritisk at vi tenker gjennom hendelsesflyten i et slikt system.

Hendelsene i diskrete hendelsessimuleringer kan organiseres på ulike måter. Dette blir ofte referert til som ulike *verdenssyn*. Man kan la hver enkelt hendelse "være sin egen herre", såkalt *hendelsesplanlegging* (*Event scheduling*), eller man kan gruppere hendelser i sekvenser ut fra hvilke prosess de hører logisk hjemme i, såkalt *prosessvekselvirkning* (*Process interaction*). En siste tradisjonell måte å organisere hendelsene på er innen hvilken aktivitet de hører hjemme i, gjerne kalt *aktivitetsinterrogering* (*Activity scanning*).

Som et eksempel for de kommende avsnittene vil vi ta utgangspunkt i en velkjent problemstilling innen emnet, nemlig bilvasken.[1] Vi tenker oss en bilvask med en innkjørsel, bilkø, vaskemaskin, noen vaskemenn, et pauserom og en utkjørsel. Biler ankommer med ujevne mellomrom og stiller seg i en eventuell vaskekø før de blir vasket og kjører videre ut utkjørselen. Vi tenker oss at vi har en mann som betjener bilvasken og holder til på et pauserom når bilvasken er ledig. Figur 2.1 viser et tenkt hendelsesforløp i modellen. Vi vil bruke dette eksempelet for å utdype forskjellene på de ulike modellene for diskret hendelsessimulering.

[1] Bilvasken viser seg å være selveste skolebokeksempelet på diskret hendelsessimulering. Vi finner den brukt som eksempler i forbindelse med blant annet Garrido[12] og Helsgaun[13] sine simulasjonssystemer.

1. *Bil1* ankommer bilvasken
2. *Bil1* inn i *vaskemaskinen*
3. *Vaskemann* ut fra pauserommet og inn i *vaskemaskinen*
4. *Bil2* ankommer bilvasken
5. *Bil2* stiller seg i vaskekø
6. *Bil1* ut av *vaskemaskinen*
7. *Bil2* inn i *vaskemaskinen*
8. *Bil2* ut av *vaskemaskinen*
9. *Vaskemann* ut av *vaskemaskinen* og inn i pauserommet
10. *Vaskemaskinen* blir uvirksom

Figur 2.1: Hendelser i en bilvask

2.3.1 Hendelsesplanlegging

Den enkleste tilnærmelsen opererer med interaksjon på hendelsesnivå, ofte kalt hendelsesplanlegging. En hendelse er der selvstendig og i løpet av sin levetid endrer systemets tilstand. I tillegg kan hver hendelse legge en eller flere ny(e) hendelse(r) i heldelseskøen, planlegge nye hendelser om man vil. Slik strider simuleringen frem helt til vi slepper opp for tid eller hendelseskøen er tom.

Hendelsen "*Bil1* ankommer *bilvasken*" kan føre til at enten "*Bil1* stiller seg i *vaskekø*" eller, dersom bilvasken er uvirksom og køen tom "*Bil1* inn i *vaskemaskinen*", men den vil uansett føre til at en ny bil ankommer - "*Bil2* ankommer *bilvasken*" - for å drive simuleringen fremover. Etter "*Bil1* ut av *vaskemaskinen*" kan enten "*Vaskemaskinen* blir uvirksom" eller "*Bil1* inn i *vaskemaskinen*" hende.

Ifølge Helsgaun[13] er dette den vanligst implementeringen av diskrete hendelsessimuleringer, da denne lar seg greit implementere uten objektorienterete programmeringsspråk. Vi ser at enhver hendelse må ta hensyn til hele systemets tilstand for å kunne bestille nye hendelser. Helsgaun har implementert en simulator basert på hendelsesbestilling i sin simuleringspakke javasimulation, men tilsynelatende mest av pedagogiske hensyn. Hans hovedfokus er på det neste verdenssynet på listen vår; *prosess-vekselvirkning.*

2.3.2 Prosessvekselvirkning

Prosessvekselvirkning er den mest klassisk objektorienterte av tilnærmingene vi tar for oss. Her sees modellen som en interaksjon mellom de ulike *prosessene* som inngår. Dette kan sees på som en innkapsling av hendelser, slik at de blir gruppert ut i fra hvem som "utfører" dem. Hver prosess er spesifisert ved en sekvens av operasjoner den kan inngå i. Det er vanlig å la en en simulasjonsleder aktivere og passivisere prosessene ut fra en tidskø.

En bilvask-simulering kan bestå av fire ulike typer prosesser; *biler*, en *vaskemaskin*, en *vaskemann* og en *innkjørsel*. Eksempelet vårt vil kunne deles opp slik:

Inkjørselen inngår i 1 og 4.
Bil1 inngår i 1,2 og 6.
Bil2 inngår i 4,5,7 og 8.
Vaskemaskinen inngår i 3,4,7,8,9 og 10.
Vaskemannen inngår i 3 og 9.

Dessuten kan moderprosessen, selve bilvasksimuleringen, sees på som en prosess og man oppnår hierarkiske modeller. (Jfr Zeiglers DEVS 2.2.2 på side 4)

Helsgaun[13] argumenterer for at en sådan prosessorientert tilnærming ofte er lett å forstå siden *prosessene* som inngår i den ofte ligner på prosesser i systemene man vil modellere, men implementasjonen av den er vanskelig.

2.3.3 Aktivitetsinterrogering

En tredje måte å gruppere hendelsene på er ved såkalt aktivitetsinterrogering. Her brytes modellen ned til de *aktiviteter* som inngår i simuleringen. En aktivitetssekvens startes bare dersom dens betingelser er oppfylt. Systemets tid springer fra hendelse til hendelse og ved hver slik blir betingelsene for aktivitetssekvenser testet. De aktivitetssekvenser som har betingelsene sine oppfylt kjøres og systemet endrer tilstand slik at andre aktiviteter kan få betingelsene sine oppfylt. Slik strider simuleringen frem helt til ingen lenger har betingelsene sine oppfylt, eller vi har brukt den simulasjonstiden vi skulle.

I eksempelet over vil vi for eksempel tenke oss en aktivitet som involverer *Bil1* mellom hendelsen "*Bil1* stiller seg i vaskekø" og "*Bil1* inn i vaskemaskin" som vi kan kalle "Vente i kø".

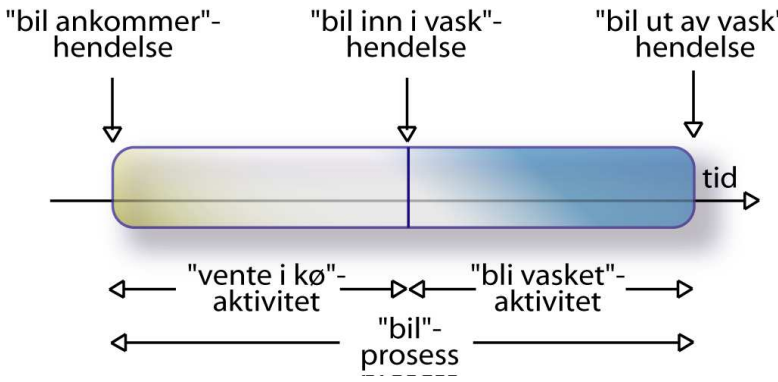
Et eksempel på implementasjoner av aktivitetsinterrogerings-verdenssynet er Grace Au[3] sin MacGrase [sic], som jeg kommer nærmere inn på siden.

Vi gjør oppmerksom på at aktivitets-interrogering noen ganger betegner en diskret *tids*simulering. Der lar vi simuleringsmotoren med jevne mellomrom spørre aktivitetene i systemet om de har betingelsene sine oppfylt, hvorpå de eventuelt kjøres.[2]

2.3.4 Skjematisk fremstilling

Figur 2.2 viser en skjematisk oversikt over hvordan tiden flyter i forhold til gruppering av hendelser i de ulike tankesettene. Denne viser en bil sin ferd gjennom vaskemaskinsimulatoren. Vi ser at hendelser er punkter i tiden, aktiviteter er tiden mellom dem og prosesser er "alt" det bilen gjør i simulasjonen.

[2]Denne tilnærmingen har en del mangler som for eksempel at man ikke kan ha aktiviteter som bruker mindre tid enn minste inkrementet i tid. Systemet kan dessuten bli unødvendig tregt da det må teste betingelsene til aktivitetene selv om ingen hendelser har oppstått. På den annen side kan man bygge mer fleksible modeller med tanke på interaksjon mellom to systemer, men dette er utenfor omfanget til denne oppgaven.



Figur 2.2: Skjematisk fremstilling over de ulike verdenssynene.

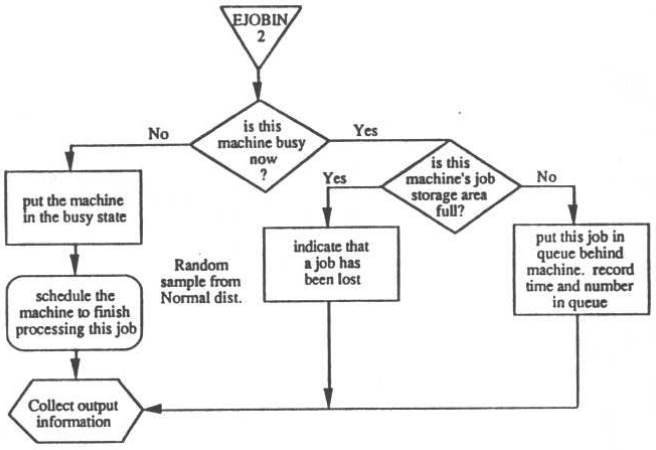
2.4 Hvilket verdenssyn egner seg best til en grafisk fremstilling av objektinteraksjon?

Så vil vi se på hvilke objektinteraksjonstankegang som passer best for å jobbe med vår problemstilling. Vi ønsker altså å kunne visualisere modellene våre. Dette innbærer at logikken i modellen skal kunne defineres reint grafisk. Vi ønsker å gjøre dette ved å dekonstruere simuleringsmodeller til generelle byggeklosser som kan knyttes sammen av buer til vidt forskjellige modeller.

Det må understrekes at *hendelsene* i en gitt modell, og da deres innbyrdes rekkefølge under simulering, vil og skal alltid være den samme uansett hvilke verdenssyn vi velger. (Fishman[9]) Grunnene til å gruppere dem på ulike måter er reint modelleringstekniske. Vi vil representere logikken i systemet på ulike vis. Det som riktignok kan hende er at vi velger ulike tilnærminger til samme problemstilling gitt ulike verdenssyn.

2.4.1 Hendelsesplanlegging

Ulempen med hendelsesplanlegging er at det fører til svært mange ulike objekter som er vanskelig å generalisere, og dermed vanskelig å kunne spesifisere/visualisere grafisk, da hver enkelt hendelse sine rutiner kan bli ganske komplekse. Se figur 2.3 for et eksempel på struktur internt i en hendelse i GASP. (Se 2.6.3 for mer om GASP.) Paul et al[21] argumenterer for at hendelsesplanlegging fører til at brukeren av simulasjonssystemet må *programmere* selv for å kunne skape modeller.



(Illustrasjonen er skannet fra [20].)

Figur 2.3: Logisk struktur i en hendelse i GASP.

Et slikt system er heller ikke særlig objektorientert da enhver hendelse må ha oversikt over vilkårlig store deler av systemets tilstand for å kunne vite hvilke hendelser det skal bestille. Dermed får en ikke nyttet de muligheter som Java tilbyr av for eksempel datainnkapsling. (Se kapittel 3 for en utdypning av hvilke fordeler objektorientering har for simulering.)

2.4.2 Prosessvekselvirkning

Den kanskje mest objektorienterte varianten er prosess-vekselvirkning, men ulempen med den er at det gjerne er flere knytninger mellom objekter som skal "samarbeide" i en modell. Objektene blir dermed vanskelig å spesifisere med tanke på generelle modelleringsobjekter. Paul et al[21] argumenterer for at simulasjonslederen må være unødvendig kompleks i sammenheng med dette verdenssynet.

2.4.3 Aktivitetsinterrogering

Det som er appellerende med en aktivitetsinterrogeringstilnærming til vår problemstilling, er at vi bare trenger å sette opp betingelsene for hver aktivitetsstart og hvordan aktiviteten påvirker systemets tilstand i stedet for å spesifisere sekvensen av aktiviteter en enhet vil bevege seg gjennom, som man må i både prosess-prosess- og hendelse-hendelse-interaksjonsbaserte systemer. Dette poenget gjør Pidd[24] i sin artikkel *Object Orientation, Discrete Simulation and the Three-Phase Approach* der han ser på en tilnærming til aktivitetsinterrogering.

Vi velger på grunnlag av disse observasjonene aktivitetsinterrogering som utgangspunkt for vårt simuleringssystem.

2.5 Tilnærminger til aktivitetsinterrogering

Pidd[24] benytter en variant av aktivitetsinterrogering kalt tre-fase-tilnærming.

2.5.1 Tre-fase-tilnærming

Tre-fase-tilnærmingen ble lansert av Tocher i 1963. Den kan i korte trekk brytes ned til følgende tre faser:

- *Fase 1:* Finn ut når neste aktivitet i modellen er for tur. Flytt simulasjonstiden til dette punktet.
- *Fase 2:* Kjør aktivitetene som ble identifisert i fase 1.
- *Fase 3:* Prøv alle aktivitetene i modellen som er kondisjonale - kjør dem som har oppfylte betingelser.

Gjenta til simulasjonen er fullført - feks. dersom en terminerende tilstand er nådd.

Et sentralt trekk ved denne tilnærmingen er at den, inspirert av prosessinteraksjonstilnærmingen tenker seg at man kan gruppere aktiviteter i B'er, som *må* hende("*b*ound to happen") og C'er, som kan skje ("happens *c*onditionally"). Et eksempel på en C kan være at en vaskemann forlater pauserommet - *som følge av noe*, mens en B kan være at en bil ankommer systemet - *bare som en funksjon av tiden.*

2.5.2 Argumenter mot trefase-tilnærmingen

Paul et al[21] kommer med tre hovedankpunkter mot trefase-tilnærmingen:

1. Enheter beveger seg fra aktivitet til aktivitet via køer. Dermed støtter den ikke deres begreper om *Ikke-ventenoder*[3] - aktiviteter som ikke bruker tid.
2. Tilnærmingen spesifiserer ingen måte for å støtte objektorientert programmering.
3. Man blir tvunget til å benytte ACD (*Activity Cycle Diagram*[4]) (eller tilsvarende diagrammer) til å spesifisere modeller.

Ut fra denne argumentasjonen mener de at det vil være vanskelig å spesifisere kompleks systemoppførsel ved trefase-tilnærmingen og lanserer fire-fase-tilnærmingen i 1997.

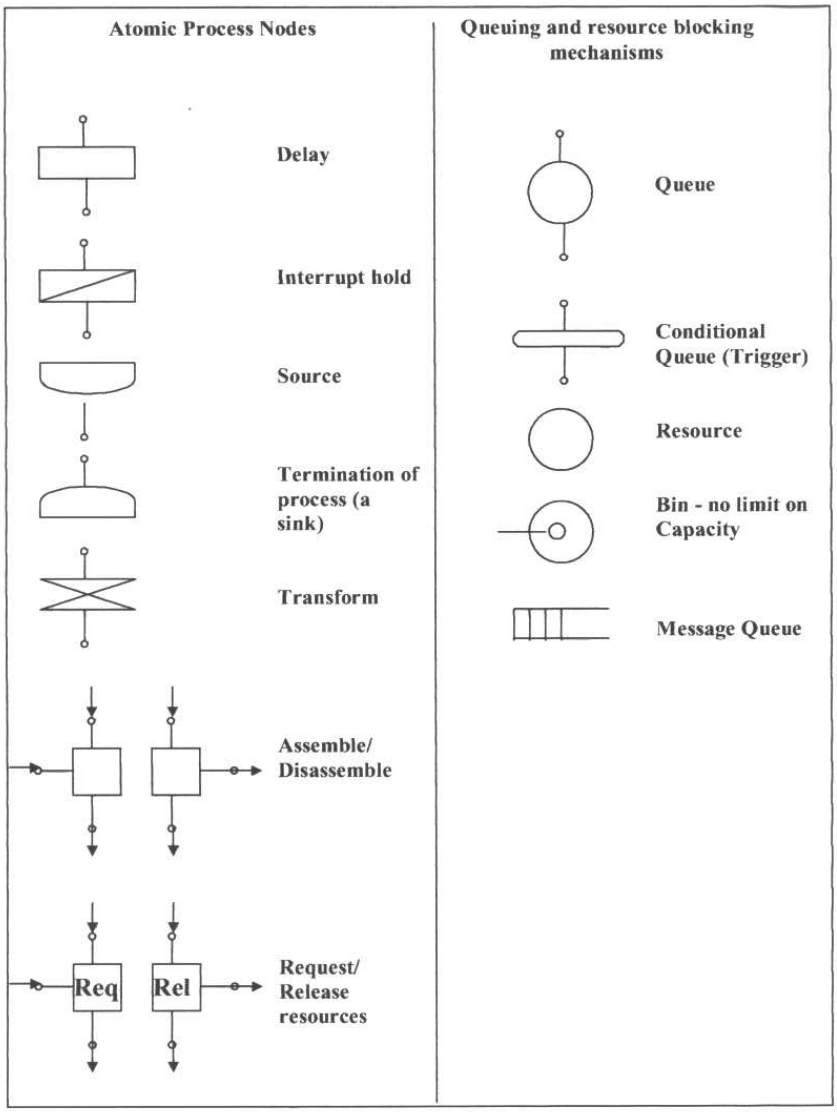
2.5.3 Fire-fase-tilnærming

Fire-fase-tilnærming (*4FT*) bygges opp rundt "Hierarkisk aktivitets-sykeldiagram"-symbolsettet[5] (*H-ACD*). Figur 2.4 viser dette. Tilnærmingen skiller mellom

[3]Begrepet *node* er bruker vi her for å understreke at modellene sees på som grafer, bestående av noder - aktivitetene - og kanter - forbindelsene mellom dem.

[4]ACD ble ifølge [21] brukt av Tocher i 1963 og Pidd i 1998 for å spesifisere modeller ved trefasetilnærmingen. Diagrammene består av to komponenter; aktiviteter og køer.

[5]Utviklet av Kienbaum og Paul i 1994.



(Illustrasjonen er skannet fra [22].)

Figur 2.4: H-ACD

ventenoder (*"Delay Nodes"*) som bruker simulasjonstid på å fullføres og såkalte *ikke-ventenoder* (*"UnDelay Nodes"*) som ikke bruker simulasjonstid.

De skisserer følgende faser simulasjonssystemet skal gå gjennom ved hver iterasjon:

- *Fase 1:* Let i aktivitetslisten og finn aktiviteten som er satt til å fullføre først. Sett simulasjonsklokken til dens stopptid.
- *Fase 2:* Stopp de aktivitetene i Ventenoder som er fullført ved aktuell tid.
- *Fase 3:* Prøv alle Ikke-ventenoder og se om noen av dem har sine betingelser oppfylt og dermed kan kjøres. Gjenta til ingen Ikke-Ventenoder kan startes.
- *Fase 4:* Kjør de relevante Ventenodene, kalkuler tid for fullførelse.

Gjenta til ingen Ventenoder står for tur til å fullføres.

Firefase-tilnærmingen finnes implementert av Paul et al[23] i JIVEsim.

2.5.4 Vår trefase-tilnærming

Ved å se på argumentene mot trefase-tilnærmingen fra 2.5.2 legger vi merke til at ved å endre litt på den kan vi muligens omgå (eller i det minste redusere betydningen av) noen av dem.

Problemet med mangel på ikke-ventenoder

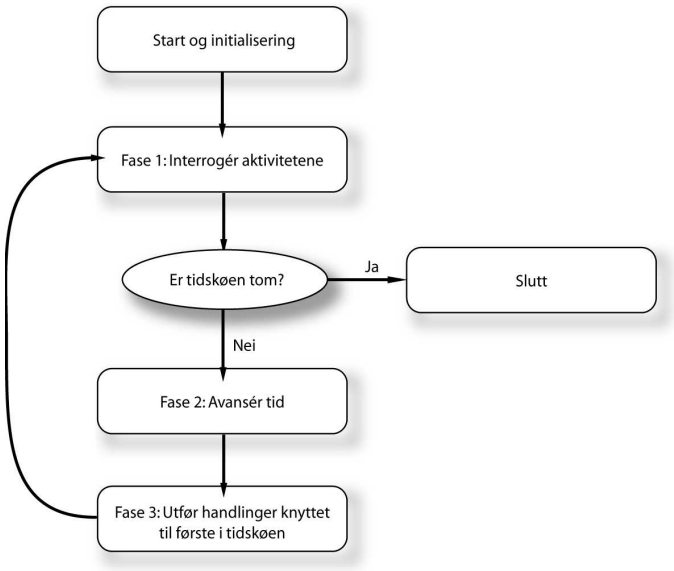
Det at en aktivitet *må* bruke tid, at vi ikke kan ha *ikke-ventenoder* kan omgås ved å gjøre et par grep.

Vi henter noen redskaper fra prosessinteraksjonstilnærmingen. Som i tradisjonell trefasetilnærming skiller vi mellom noder som er kondisjonale og de som ikke er det. Vi kaller dem *aktiviteter* (kondisjonale noder) og *prosesser* (ikke-kondisjonale noder). Dermed kan vi holde en liste med noder som *må* skje i en hendelseskø sortert etter når de kommer til å endre systemets tilstand, som i prosess-vekselvirkningstilnærmingen. Noder som *kan* skje holdes i en annen liste. Når en node har fått betingelsene sine oppfylt, flyttes den til listen over noder som må skje, altså hendelseskøen. Da gjør aktiviteten det den er satt til å gjøre og etter det *må* nemlig aktiviteten bli ferdig med det. Da blir fasene i vår tilnærming som følger:

- *Fase 1*: Gå gjennom listen med aktiviteter og flytt dem som har betingelsene sine oppfylt til tidskøen (bestill deres ferdigstillelse).
- *Fase 2*: Sett simulasjonstiden til tiden da den første aktiviteten i tidskøen står for tur.
- *Fase* 3: Aktivér den noden som er for tur i tidskøen og la den gjøre det den er satt til.

Dette gjentas helt til hendelseskøen er tom eller at vi har sluppet opp for simulasjonstid. Figur 2.5.4 viser dette skjematisk.

Siden vi ikke "flytter" simulasjonstiden før vi har sjekket om noen av de kondisjonale nodene har betingelsene sine oppfylt, kan godt disse operere med tider for ferdigstillelse på 0 tid. Da blir de bare rett og slett sidestilt med



Figur 2.5: Vår tilnærming til aktivitetsinterrogering

eventuelle ikke-kondisjonale noder, som *Ikke-ventenodene* til firefasetilnærmingen. Dette fører da også til at vi kan lage disse nye former for noder som firefasetilnærmingen introduserer.

Problemet med mangel på objektorientering

Pidd viser i sin artikkel kalt *Object-orientation, Discrete Simulation and the Three-Phase Approach*[24], at en forening av tre-fase-tilnærmingen og objektorientering er fult mulig.

Vi implementerer ikke objektorienteringen som Pidd, men likefullt objektorientert. Hver node i grafen som spesifiserer en modell - for å bruke terminologien til Paul et al - er et objekt som enten er en aktivitet eller en kø[6]. Disse har generiske grensesnitt for enhetsflyt - køer har en inn- og en ut-port, aktiviteter har diverse inn- og ut-porter. Vi kommer nærmere inn på detaljer rundt dette i kapittel 6.1.

Problemet med at vi må bruke ACD-diagrammer

Siden vi har en mulig løsning på problemet med ikke-ventenoder, gjør dette at vi kan bruke et ikonsett svarende til H-ACD for å spesifisere modeller. Vi vil komme tilbake til dette i kapittel 5.

[6]Nodene kan også være spesialobjektene fordeling eller enhet, men disse inngår ikke i selve nettverket av enhetsflyt.

2.6 Hvordan andre har implementert diskret hendelsessimulering

Det finnes allerede en del implementasjoner av diskret hendelsessimulering. Både for å sette dem i et historisk perspektiv, men også for å låne idéer fra dem og vise hva som skiller vår tilnærming fra allerede eksisterende.

2.6.1 Generelle programmeringsspråk for simulering

Artikkelen *A History of Discrete Event Simulation Programming Languages* av Nance[20] forteller om fremveksten av ulike redskaper for diskret hendelsessimulering.

GPSS

Serien *General Purpose System Simulator* (GPSS) ble påbegynt i 1960 og var det mest populære språket for simulering i de såkalte "tidlige dager for simulering"[7].[20] Grunnen til dette var nok i stor grad at det var utviklet på IBM[8] sine maskiner i de dager og dermed ble pakken for diskret hendelsessimulering for disse markedsdominerende maskinene. Det at selv studenter som ikke studerte data kjapt kunne konstruere modeller[9] gjorde også at den kunne brukes i utdanning, så vel som forskning, noe som er et annet vesentlig bidrag til dens utbredelse.

Språket bruker en semi-grafisk notasjon for kommandoer. Disse inngår så i blokkdiagrammer som spesifiserer modellenes logikk. Dette er basert på prosessinteraksjon.

Det som gjør dette systemet spesielt interressant for oss er at det er det første simuleringssystemet som ble koblet til en interaktiv terminal for sanntids observasjon av modeller under kjøring av simuleringen.

GPSS er ikke et prosedyrespråk, så det kan ikke spesifisere like komplekse systemer som andre språk, som feks. Simscript, kan.

Simscript

Simscript er en simuleringspakke i form av et prosedyrespråk implementert av Harry Markowitz et al.[20] Det er laget for store diskrete hendelsessimuleringer. Første versjonen kom i 1963 og var en preprosessor til Fortran. Verdenssynet er basert på hendelsesplanlegging og bygget opp av enheter, attributter og sett. To år seinere kom Simscript I.5 som gir maskinkode og som også tillater et prosessorientert verdenssyn.

Med Simscript II.5 kan man bl.a. bruke markov-kjeder[10] til å bygge opp modeller man vil simulere. Det er dessuten ment som et generelt programmeringsspråk, i motsetning til I som var myntet på simulering. En ting som er verdt

[7] Perioden 1960-1975 regnes ofte som *"de tidlige dager for simulering"*.[20]

[8] *Store Blå* var som kjent markedsleder innen dataindustrien da.

[9] Nance utdyper i [20]: *"although the logic switches might have puzzled many business students"*

[10] En annen fascinerende bruk av markov-kjeder er å kategorisere musikk. (Yi-Wen Liu, *"Modeling music as Markov chains - composer identification"*, http://ccrma.stanford.edu/~jacobliu/254report/)

å bite seg merke i er at man skiller mellom enheter som er midlertidige og enheter som er permanente i systemet. Dette er et skille vi også vil benytte i våre simuleringer.

Simscript er en viktig inspirasjonskildene til Simula.

Simula

Simula I ble implementert i 1964 av Ole-Johan Dahl og Kristen Nygaard og regnes som det første objektorienterte programmeringsspråket.[11] Språket er en utvidelse av Algol 60 for diskret simulering. Det de prøvde å oppnå var å lage et system for å spesifisere komplekse systemer med den hensikt å simulere dem. Løsningen deres viste seg å kunne brukes til å kunne beskrive komplekse systemer generelt, så Simula endte opp som et generelt programmeringsspråk. Ved å innføre begrepet *record class* ledet de vei til dataabstrakson og dermed objektorientert programmering generelt. Simula67 er oppfølgeren og kom i 1967.[35, 7]

En annen idé som ble lansert i Simula I var *korutinene* og deres kvasiparallellitet. Disse er forfedrene til det vi i dag har i Java som tråder.

2.6.2 Java-baserte

Som vi kommer inn på i kapittel 3 er Java et egnet språk å lage simuleringssystemer i.

javasimulation

javasimulation er en simuleringspakke for java utviklet av Keld Helsgaun[13]. Den støtter alle de tre tilnærmelsene til objektinteraksjon vi har vist over, men hendelsesplanlegging og aktivitetsinterrogering er knapt vektlagt og mest med som eksempler. Fokuset er på prosessinteraksjon.

Det spesielle med Helsgaun sin pakke, javasimulation, er at det er tilrettelagt for simula-brukere. Man programmerer i "rein" java, men han har utviklet en syntaks som minner om den man finner i SIMULA. Dette for å lette overgangen for programutviklere med bakgrunn fra nettopp dette språket, men også for å bygge på denne velutprøvde måten å modellere på. Han har gjort dette ved bla å opprette en del syntaktiske *konstanter*.

For eksempel kan en skrive følgende i Simula:

```
activate bil at 42 prior;
```

Helsgaun har innført objektene *at* og *prior*, og vi kan kalle tilsvarende rutine i Java slik:

```
activate(bil, at, 42, prior);
```

Ved å la disse konstantene være av ulike klasser vil man kunne oppdage feil bruk av dem under kompilering av koden.

[11] Dahl og Nygaard fikk Turing-prisen i 2001 "for fundamentale idéer for fremveksten av objektorientert programmering, gjennom deres design av programmeringsspråkene Simula I og Simula 67". Prisen omtales noen ganger som "Nobel-prisen i informatikk" og er oppkalt etter Alan M. Turing som ofte regnes som datavitenskapens far. Tidligere prismottakere er vitenskapsmenn som Dijkstra (1970), Knuth (1974), Rivest, Shamir og Adelman (2002).

Dessuten har han laget biblioteker som inneholder rutiner og datastrukturer som finnes i simula, men ikke Java. Spesielt har han implementert korutinene fra Simula ved hjelp av tråder i Java. Derfor er simulasjonspakken også trådbasert (et begrep som utdypes nærmere i kapittel 3).

PSim-J

Prosessinteraksjon er også verdenssynet som simuleringspakken til Garrido[12]; PSim-J, benytter seg av. Pakken har han oversatt fra sin egen PSim, som er skrevet i C++.

simJava

simJava er en diskret hendelses-simuleringspakke for Java oversatt fra C++ (HASE++ av Fred Howell[37]). Denne baserer seg på at bruker programmerer modeller i java ved hjelp av medfølgende biblioteker. Under kjøring kan man benytte innebygde rutiner for å sette opp et grafisk grensesnitt der en kan endre på parametre til modellen uten å måtte kompilere systemet om igjen. Det er også muligheter for animerte tilbakemeldinger. Disse to sistnevnte egenskapene gjør systemet svært egnet til nettleserbasert simulering. Som i javasimulation er simulasjonsmoteren trådbasert og implementerer prosess-prosess-interaksjon.

bergmann.simulation

Ørjan Bergmann[6] har laget et generelt rammeverk for simulering (bergmann.simulation). Dette er basert prosessinteraksjon og svært generelt implementert. Det er ikke trådbasert. Bergmann viser hvordan systemet hans kan brukes til å gjøre eksperimenter på navigasjonsevnene til rotter, men også en klassisk heissimulering.

JIVEsim

JIVEsim (*Java Iconic Visual Environment for Simulation)[23]* er en pakke utviklet av Odhabi, Paul og Macredie i Java. Den bruker fire-fase-tilnærmingen til Paul og utvikler et ikonspråk for å sette opp systemet grafisk ved hjelp av ikoner og arker.

Systemet genererer Java-kode som man kan finkode for så å kjøre. Dette medfører at om en vil endre på parametre til systemet må man starte systemet på nytt.

JavaBeans

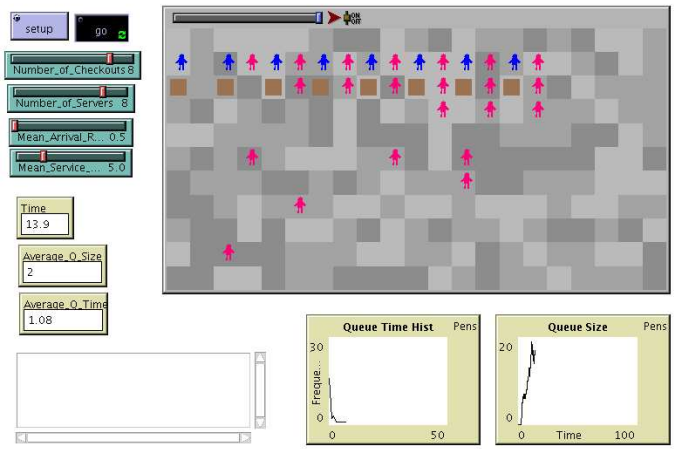
Prähofer et al [25, 26, 28] argumenterer for å bruke JavaBeans sin komponentorienterte tankegang som implemetasjonsbase for DEVS-inspirerte modeller. Dermed kan man benytte seg av standard Bean-byggeredskap. SimBeans er en pakke for diskret hendelsessimulering implementert slik.

NetLogo

Netlogo[12] er et fritt tilgjengelig generelt simuleringssverktøy implementert i Java av Uri Willenski. Man programmerer modeller i et språk basert på LOGO. Dette

[12] http://ccl.northwestern.edu/netlogo/

gjør verktøyet svært generelt og man kan løse mange ulike simuleringer. Figur 2.6 viser en købasert modell for kasser i et supermarked. Fokuset er i høy grad på grafiske tilbakemeldinger i form av animasjoner. Modellene kan lagres som appleter for mulig innkorporering i nettsider.



Figur 2.6: Visuell tilbakemelding fra en simulering av supermarked-kasser i NetLogo.

2.6.3 Simuleringspakker implementert i andre språk

MacGraSE

MacGraSE (*Machintosh Graphical Simulation Environment*) er en pakke for å grafisk modellere simuleringer i et brukervenlig grensesnitt utviklet av Grace Au[3, 4]. Denne er basert på aktivitetsinterrogering og ACD-diagrammer.

MacGraSE er veldig grafisk orientert. Det lar en koble opp modeller ved ikoner og oppfordrer til å bruke et bilde av det du vil modellere som en kulisse for modelleringen din. Dette for å forenkle den kognitive lasten til en bruker i et typisk bruker-spesialist samarbeid.

Det grafiske grensesnittet genererer kode som en kan finkode for å spesifisere et mer komplekst system enn det grafiske tillater, før man kjører simuleringen.

SimPack

Simuleringspakken SimPack er utviklet av Paul Fishwick og skrevet i C (og tildels C++). Pakken er fritt tilgjengelig på verdensveven. Målet med denne er å gi forskere og studenter et sted å starte med simulering. En nyere variant er Sim++ med en mer objektorientert design.

GASP

GASP*(General Activity Simulation Program)* er et sett av subrutiner for Fortran utviklet for simulering av Philip J. Kiviat på tidlig 60-tall.[12]

2.6.4 Hvorfor vi implementerer simuleringsverktøyet helt forfra

Selv om vi har alle disse allerede eksisterende pakkene for simulering velger vi likevel å implementere vår egen. Dette er det flere grunner til:

- Vi vil eksperimentere med hendelsesflyten. Ingen av disse lar oss eksperimentere med hvordan selve hendelsesflyten i systemene går. Vi vil vise at en modifisert trefasetilnærming kan benyttes i et system for købasert diskret hendelsessimulering.
- Vi vil dermed eksperimentere med å bygge modeller i et grafisk grensesnitt og kjøre dem på sparket. Dette er unikt for mjmod i forhold til de vi har sett på her. Ved å gjøre en slik vri er det veldig fort gjort å gjøre endringer både i parametre og implementasjon uten å måtte gå til en kildekode for endringer.
- Java har en god del egenskaper og kommer med en del redskaper som passer til diskret hendelsessimulering.

Dette siste punktet vil vi utdype i neste kapittel.

Kapittel 3

Java og diskret hendelsessimulering

I dette kapittelet ser vi på relevante aspekter ved Java i forhold til diskret hendelsessimulering.

3.1 Egenskaper ved objektorientert programmering egnet for diskret hendelsessimulering

Java er et objektorientert programmeringsspråk. Et indisium på at diskret hendelsessimulering passer godt overens med det objektorientert paradigmet for programmering, kan vi finne i historien bak Simula. Dette første objektorienterte språket var jo nettopp utviklet for å studere diskret hendelsessimulering. Der er klassebegrepet sterkt knyttet til et mønster for prosessinteraksjon. Dahl og Nygaard så for seg at en kunne sette opp modeller ved å spesifisere hvordan prosesser samhandler.

3.1.1 Arv og polimorfi

Arv og polimorfi er de mest brukte begreper i forbindelse med objektorientert programmering, så vi skal ikke dvele ved dem her i oppgaven. Kort forklart: Arv (*inheritence)* er et greit redskap for å lage systemer generelt, da det lar oss bruke kode om igjen. Polimorfi bruker vi i utstrakt grad for å la en og samme rutine jobbe på mange ulike typer objekter med samme stamfar.

3.1.2 Abstraksjon

Java gir oss muligheten til å gjemme unna uønskede detaljer fra kode - ofte kalt abstraksjon (*abstraction*). Man innså tidlig nytteverdien i å samle dataene og operatorene som jobber på dem. Ravi Sethi[30] poengterer at abstraksjon ble beskrevet av Wilkes, Wheeler og Gill i deres samling subrutiner så tidlig som i 1951.

> "When a programme has been made from a set of sub-routines the breakdown of the code is more complete than it otherwise be.

> This allows the coder to concentrate on one section of a programme at a time without the overall detailded programme continually intruding." (Wheeler, 1952)

Dette er altså ikke spesielt for objektorienterte språk, men en strengere form for abstraksjon som finnes i objektorienterte språk er såkalt innkapsling.

Innkapsling

Innkapsling (*encapsulation*) vil si at man lager klasser med så få allment tilgjengelige strukturer (data og operatorer) som mulig. Man hindrer rett og slett andre objekter tilgang til rutinene. Dette medfører at objekter gjør at programmer blir enklere å lese og vedlikeholde.

Innkapsling er et kraftig verktøy for å skape komponentbaserte systemer.

3.2 Biblioteker i Java

Utviklingsmiljøet til Java kommer med en temmelig stor standard-API, men selv om Java kommer med et stadig større standardbibliotek[1] dekker det ikke direkte diskret hendelsessimulering (enda).

3.2.1 Redskaper for diskret hendelsessimulering

Fishman[9] har satt opp en liste over hva et programmeringsspråk for diskret hendelsessimulering må kunne:

- Kontinuerlig opprettholdele en liste med de hendelser som er planlagt å kjøre.
- Opprettholde dynamiske datastrukturer.
- Generere pseudotilfeldige tall.
- Generere utvalg av tilfeldige tall fra forskjellige fordelinger.
- Regne ut sammendragsstatistikk som gjennomsnittstider, varianser og histogrammer.
- Generere rapporter.
- Finne feil som er spesielle for simulasjonsprogrammer.

Java sine standardbiblioteker innehar allerede en rekke av disse egenskapene.

De har støtte for sorterte samlinger med objekter som kan man bruke til å løse første kriteriet på Fishman sin liste. Dynamiske datastrukturer finnes også i flere varianter, som for eksempel lenkede lister. Klassen java.util.Random kan brukes til å generere pseudotilfeldige tall. Klassen java.lang.Math har de fleste redskaper for å regne ut sammendragsstatistikk. Java har funksjoner i biblioteket som følger med for å generere rapporter både i form av tekst og grafikk.

[1] API til Java tilgjengelig på: http://java.sun.com/j2se/1.4.2/docs/api/ eller bedre i *Java in a Nutshell* av David Flanagan[11].

3.2.2 Redskaper for å skape et grafisk brukergrensesnitt

I Java sine såkalte fundamentale klasser (*Foundation Classes*) finnes redskaper for å bygge opp grafiske grensesnitt; *Active Window Toolkit* (*AWT*) og *Swing*.[10] Selv om disse nok ikke er like kraftig som for eksempel C++-biblioteket Qt fra Trolltech, er de enkle og oversiktlig i bruk, så man kan kjapt lage grafiske grensesnitt til sine Java-programmer.

I tillegg til dette har Java en del andre egenskaper som gjør det attraktivt å bruke til diskret hendelsessimulering.

3.3 Egenskaper ved Java egnet for diskret hendelsessimulering

3.3.1 Tråder

Java er et såkalt flertrådet programmeringsspråk. Tråder (*threads*) blir ofte referert til som *lettvekts-prosesser,* da det er nettopp det det er - enkelt sekvensiel strøm av kontroll innen et program. Parallellitet om man vil. Programsnutter trenger ikke å kjøres sekvensielt, men parallelliteter i underliggende system kan utnyttes, uten at du som programutvikler trenger å tenke (særlig) over det utover å starte subrutiner i tråder. I grafiske grensesnitt og vindusmiljøer er dette bortimot en nødvendighet. Dette for at vi skal kunne lage programmer som kan kjøre kodesnutter i bakgrunnen samtidig som det holder et grafisk grensesnitt oppdatert. Java sine biblioteker for grafikk AWT/Swing benytter seg av nettopp dette. Et hvert javaprogram vil dessuten ha tråder som feks. søppeltømmeren (*garbage collector)* som fjerner objekter fra minnet dersom de ikke lenger er i bruk, og dermed frigjør minneressurser. Disse trådene går parallellt og deler på CPU-kraften.

Tråder og diskret hendelsessimulering

Både Helsgaun[13] og Garrido[12] benytter tråder i sine implementasjoner av diskret hendelsessimulering i sine prosessvekselvirkingstilnærminger. Det de utnytter ved trådene er spesielt deres "strukturelle" kvaliteter. Nemlig det at man kan ordne prosesser i slike og dermed oppnå et ryddig system. De utnytter ingen parallellitet i sine systemer. Dermed er dette egnet til deres tilnærming, men mindre egnet for våres, da en aktivitet er enklere bygget opp enn en prosess(2.4 på side 8). Vi kan bruke bilvasken til å illustrere dette også. I en prosessvekselvirkningsvariant av denne inngår både bilprosessen og vaskeprosessen i "aktiviteten" vasking, begge vil altså fortsette sine "liv" etter denne aktiviteten og en kan ved hjelp av tråder holde styr på hvor i deres "liv" de har kommet. Man si at en aktivitet bare har én start og én stopp, mens en prosess kan gjerne starte og stoppe flere ganger i løpet av sin kjøring.

Vi kunne være fristet til å bruke tråder til mer enn struktur. Altså slik de er "ment". Problemet er at svært få (eller ingen) rutiner går i parallell i et diskret hendelsessimulasjonssystem. Naturen til diskret hendelsessystemer er jo nettopp det sekvensielle. Alle hendelsene er koblet til en og samme tidsakse. Vi vet ikke hva som vil hende like etter en aktivitet er over, da tilstanden til systemet har endret seg. Settet med betingelser for oppstart av aktiviteter - systemets

tilstand - er altså endret.[2] Dermed vil en ikke få utnyttet hovedstyrkene til tråder - parallelliteten.

Noen systemer for diskret hendelsessimulering kan man forøvrig tenke seg vil gagne en parallellisering; de systemene der det en aktivitet eller prosess skal *gjøre* er prosessorkrevende i forhold til selve simulatoren. Da vil vi ønske å la dette gjøres mens vi kan la simulasjonen gå til det simulasjonstidspunktet der dette skal være ferdigstilt. Det hele koker da ned til om aktivitetens eller prosessens rutiner har en kjøretid som er i størrelsesorden like stor eller større enn selve simulatoren. Det vanskelige her er at man må vite hvor lang tid en aktivitet eller prosess tar *før* dennes interne rutiner kan kjøres i parallell. Dette fordi vi ikke kan la simulasjonstiden gå forbi ferdigstillelsen av den, og den kan bruke vilkårlig liten tid.

Det finnes løsninger på også dette problemet. For eksempel beskriver Panajotis[3] den såkalte *"Time Warp"*-mekanismen som lar deler av simulasjonene gå i parallell. Hver gang en hendelse i en del av simulasjonen påvirker en annen del sjekkes det om diverse kausalitetsbetingelser er oppfylt. For eksempel om simulasjonstiden er synkron. Dette gjøres ved å sjekke tidsstempelet som hendelsen har. Dersom en del av simulasjonen har kommet for langt simulasjonstidsmessig har den en roll-back-funksjon som flytter simulasjonstiden slik at kausaliteten er oppfylt. I praksis må man ta vare på stadier på veien til gjeldende lokale simulasjonstid for så å forkaste store deler av dem. Dette er et vidt forskningsfelt og utenfor grensene for denne oppgaven, så vi vil ikke gå i dybden på dette.

3.3.2 Flyttbarhet

En annen grunn til at Java egner seg til komponentbaserte systemer er at det er flyttbart (*portabelt*). Dette er også medvirkende til at Java har slått seg opp samtidig med at verdensveven har bredt om seg. Det er laget for å være kompatibelt med alle operativsystemer som har implementert en virtuell maskin for å håndtere *byte-koden,* som javakildekode blir "kompilert" til, men dette er temmelig utbredt. Dette gjør at man kan utvikle kode på en plattform og det vil være kjørbart "overalt". Denne egenskapen er ikke noe man nødvendigvis utnytter ved simulering, men generelt er det en god tanke at kode man skaper ikke er avhengig av en plattform for å kjøre.

Flyttbarhet og diskret hendelsessimulering

Ulemper er det selvsagt med å ikke kompilere til kjørbare filer *nativt* i et operativsystem eller til en arkitektur men til dette mellomstadiet som skal oversettes i sanntid under kjøring. For eksempel har Java et rykte på seg til å være riktig så tregt, noe som oss som implementerer simulatorer ikke setter pris på. Vi skal jo gi programmene våre tusener av tall å knuse og problemer å knekke.

Heldigvis blir de virtuelle maskinene mer og mer effektive for hver versjon av Java Sun slepper, så gapet mellom kjøretid for binær-kode og byte-kode blir mindre og mindre. Både IBM og Sun har implementert såkalte *Just-in-time*-kompilatorer for Java for å få opp kjørehastigheten. Dette innebærer å oversette

[2] Vi kan tenke oss at vi innfører måter å definere totalt adskilte grener i modellen, som aldri vil interaktere, men da beveger vi oss bort fra idéen med en tidskø der alle hendelser henger på, og dermed farlig nær yttergrensene for diskret hendelsessimulering.

[3] Panajotis, K, *Parallell Regenerative Queueing Network Simulations,* Teknisk rapport, Aristoteles Univeristet, Tessaloniki, Hellas

byte-koden til maskinkode "på sparket" under kjøring, noe som har vist seg å være effektivt.

Den strenge objektorienteringen sammen med denne flyttbarheten har gjort det interessant å utvikle komponentbaserte systemer i Java, gjerne med såkalte JavaBeans.

3.3.3 JavaBeans

JavaBeans er en komponentarkitetur for Java som har bredt om seg. Her kan man bygge opp komplekse programvare-systemer ut fra basiskomponenter som kan være mer eller mindre skreddersydde for diverse oppgaver. Det finnes mange grafiske "bønnebyggere" på markedet, som kanskje kunne være gunstig å benytte seg av i vårt tilfelle. Da skulle man kunne lage til diverse standardkomponenter for diskret hendelsessimulering, og koble modellen sammen i en sådan bygger.

JavaBeans og diskret hendelsessimulering

SimBeans er en pakke for diskret hendelsessimulering der dette er implementert. Men, slik vi ser det er bønne-idéen i Java basert på at komponentene skal kunne kobles direkte sammen, at kontrollflyten i programmet går fra bønne til bønne, noe som skaper kluss i et strengt diskrete prosessorienterte hendelsessimuleringsmiløø. Vi vil jo ha et overordnet system som holder styr på hvilke prosesser som skal starte når - utfra tidskøen vår. SimBeans løser det ved å implementere en slags proxy-bean, en mellommann, som holder styr på hvordan simuleringen skrider frem. Alle "meldinger" fra en prosess til en annen må gjennom denne, og dermed brytes oversiktligheten i modelleringen ned. Om alle koblinger fra bønne til bønne må gå gjennom en mellommann, vil dette redusere den visulle oversikten over logikken i modellen.

Ut fra argumentasjonen over velger vi å hverken benytte oss av tråder eller *JavaBeans* i vår implementasjon.

Kapittel 4

Spesifisering av en simuleringsmodell

> *"The limits of my language mean the limits of my world."*
> Ludwig Wittgenstein

Nå har vi spesifikasjonene til simulatoren på plass. I dette kapittelet vil vi se på hvordan vi kan spesifisere modellene.

For å kunne kjøre modeller i vårt simuleringssystem må vi, som vi var inne på i avsnitt1.1.4, skape et språk for å spesifisere disse. Vi vil konstruere et språk som kan brukes for å spesifisere modeller i et grafisk grensesnitt slik at simulatoren vår kan kjøre dem.

Vi vil se på to ulike redskaper for å bygge opp et slikt språk, samt to måter å håndtere modellene for simulasjonssystemet.

4.1 Spesifisering av modell i et eksisterende språk

Å benytte et eksisterende språk for å spesifisere en modell kan være nyttig. Da kan vi bruke verktøy som allerde finnes for å jobbe med modellene vi skaper.

4.1.1 UML for spesifisering av modell

Unified Modelling Language, eller UML, er for mange industristandarden for å spesifisere, visualisere, og konstruere komponenter i programvaresystemer.[36] Dermed vil det være nærliggende å bruke dette også i vår sammenheng. Garrido bruker det i sin bok for å vise modeller, men bare for å skape "papirmodeller" før han koder modellen i java ved hjelp av PSimJ-bibliotekene.[12] Han lar ikke dette inngå i en automatisert prosess for generering av hverken kode eller kjørbare simuleringer, men bruker dem for å vise deler av strukturerene i sine modeller.

Vi ser derfor videre på et mer generelt verktøy for dataspesifisering, nemlig XML.

4.1.2 XML for spesifisering av modell

XML står for *eXtensible Markup Language*, som kan oversettes med "Utvidbart markeringsspråk"[1]. Det er et metaspråk utviklet av "The World Wide Web Consortium" (W3C). Med metaspråk mener vi grovt forklart at det er et språk for å bygge opp andre språk.

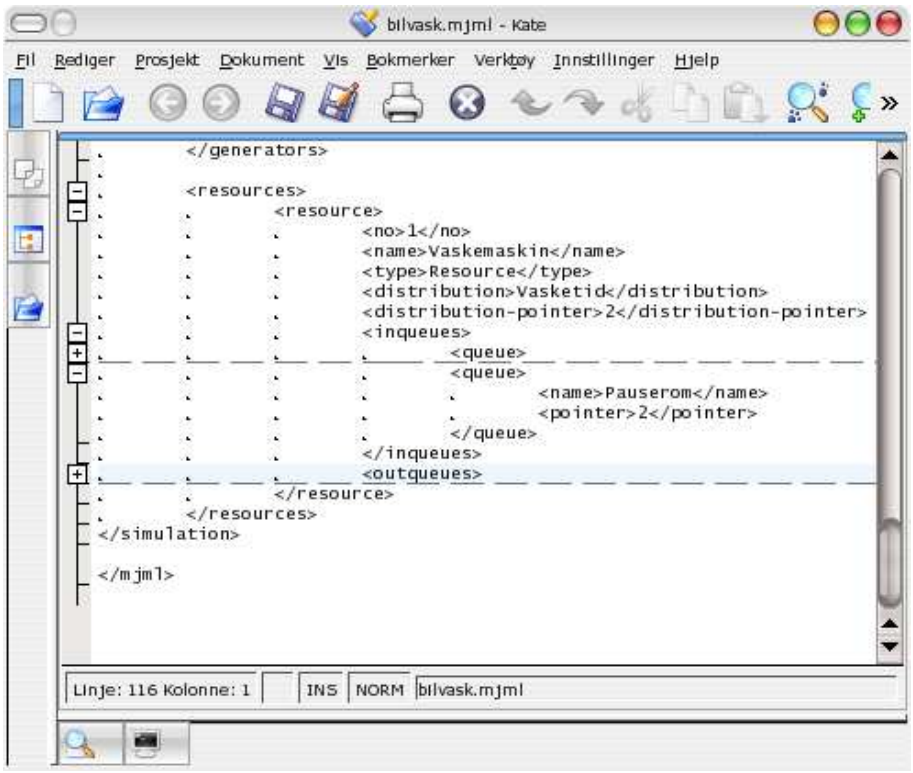
Hvorfor XML

Det at nettopp W3C står bak XML er en av hovedgrunnene til å bruke det. Formatet er basisen for flere og flere såkalte åpne filformater. Åpne i den forstand at filformatet er veldokumentert og ikke pålagt lisenser for bruk. Eksempler er *OpenOffice* sine dokumentformater[2] og *SVG* (skalerbar vektorgrafikk)[3]. Matematikere har utviklet XML-baserte MathML[4] for å skape et universelt språk for å beskrive og visualisere matematikk[5]. Chemical Markup Language (CML)[6] er et språk for å utveksle informasjon om molekyler.

XHTML er arvtageren til HTML for å definere sider på verdensveven. Dette er HTML spesifisert på i XML og dens strengere syntaks.[7] XML holder dessuten på å bli *de facto* standard for svært mange andre nettbaserte tjenester,[8] som feks. SOAP og WSP.

Det eksisterer derfor allerede flere verktøy og hjelpemidler for å sette opp jobbe med XML-filer. Man kan feks. i Kate for KDE, et fritt tilgjengelig program, nøste og syntaksmarkere XML-kode som vist her på figur 4.1.

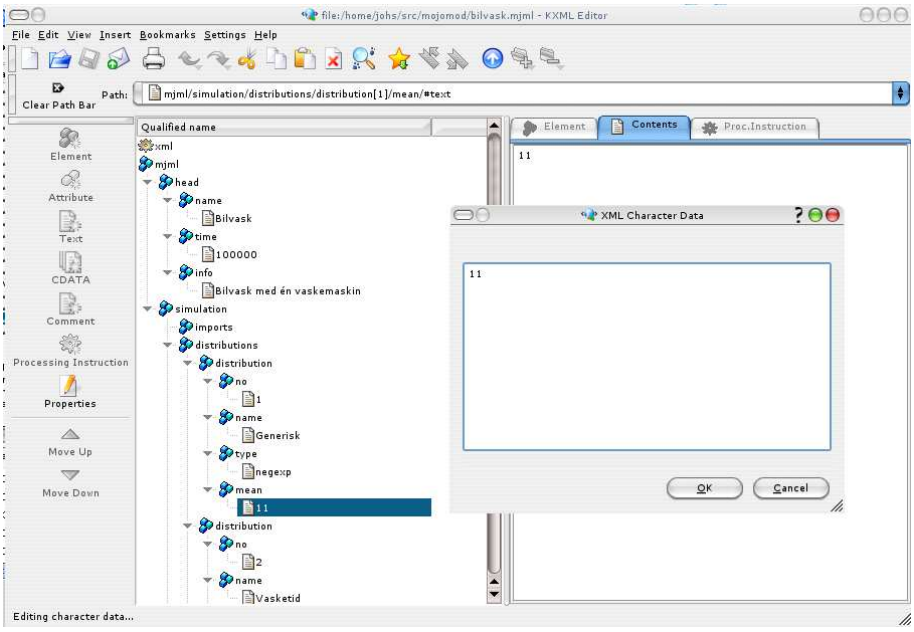
[1] Se tillegg B.4 for diskusjon rundt fornorsking av begrepet XML.
[2] http://www.openoffice.org/
[3] http://www.w3.org/TR/SVG/
[4] *W3C Math Home*, http://www.w3.org/Math/
[5] Dette er mye som Latex, men med fordelene XML medfører.
[6] *Chemical Markup Language (CML)*, http://www.xml-cml.org/
[7] De aller fleste nettlesere har støttet XHTML en tid allerede.
[8] *XML:It's the future* of HTML (http://www.sun.com/980602/xml/)



Figur 4.1: XML-redigering i Kate

Reine XML-redigeringsverktøy kan en også ty til, som feks. KXMLeditor, igjen et fritt KDE-program (Figur 4.2). Andre frie sådanne verdt å nevne er: Amaya[9] fra W3C og Cooktop[10]. Dessuten finnes det kommersielle programmer for å jobbe med XML, som feks. Altova[11], XML Mind XML Editor[12] og Exchanger XML Editor[13].

[9] http://www.w3.org/Amaya/
[10] http://www.xmlcooktop.com/
[11] http://www.altova.com/
[12] http://www.xmlmind.com/xmleditor/
[13] http://xngr.org/



Figur 4.2: XML-redigering i KXMLeditor.

XML minner på mange måter om HTML[14]. Begge språkene er delmengder av SGML. Dette bringer oss til det neste argumentet for å bruke XML; brukere som kjenner HTML vil kjapt kunne sette seg inn i et språk basert på XML. XML er nemlig vel så lesbart, eller ordrikt som HTML. En XML-fil kan være selvforklarende noe som gjør det lettere for en bruker å koble opp modeller.

XML er dessuten såpass utbyggbart at det kan benyttes i forbindelse med å koble opp grafen i et reint grafisk miljø.

4.2 Hvordan skal simulasjonssystemet sette opp en kjørbar versjon av modellen?

Vi kan løse problemet med overgangen fra en modell spesifisert i et språk til å kjøre den i hovedsak på to måter. Vi kan enten generere javakode fra en modell som så kompileres og kjøres, eller tolke (*parse*) en modell og kjøre denne på direkten.

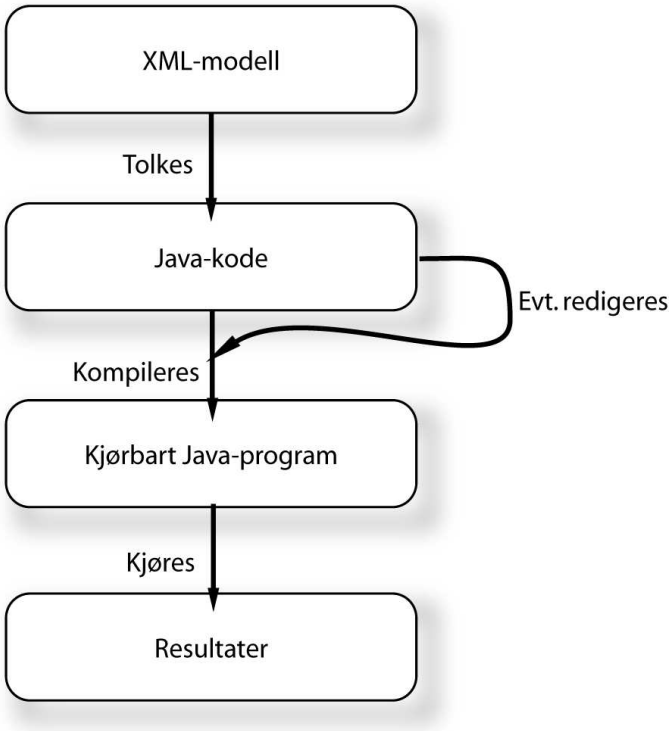
4.2.1 Generering av java-kode fra modell

En måte å løse overgangen fra modellen til kjørbar kode vil være å "oversette" modellens logiske struktur til java-kode. Deretter vil brukeren ha mulighet til å finjustere saker og ting i den genererte koden før man kan kompilere for så å kjøre programmet i en java virtuell maskin.

[14] W3C står bak både HTML, XML og SGML.

Java-kode generert fra modellspesifikasjoner

Au[3] sin MacGraSe er et eksempel på en slik tilnærming, men hun genererer riktignok ikke Java-kode.



Figur 4.3: Flyten i et program som genererer java-kode fra en XML-modell.

Fordeler

Et slikt system kan bli svært så fleksibelt, siden man kan finkode objektene selv i ettertid og har dermed hele ordforrådet til Java å boltre seg med.

Ulemper

Den største ulempen i vår sammenheng er at terskelen for brukeren høyere enn i et system som tolker modeller i sann tid, da det beror på kunnskaper i Java.

Et problem med en slik tilnærming er at det kan være vanskelig å få XML-språket til å definere annet et skall av kode som man så må så finkode selv før man kompilerer den. Dette må en muligens gjenta ved hver testkjøring. Som en løsning på dette kan vi se for oss et system som tar vare på finkodingen fra versjon til versjon, men dette igjen kan innføre problemer med inkorporering av kodesnuttene med det som blir generert ut fra XML-filen forøvrig.

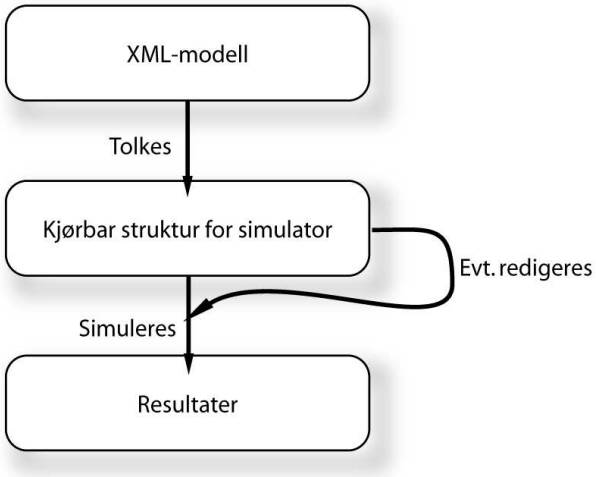
Dessuten blir en evt. visualisering av selve simuleringen vanskeligere, jo mer generell koden blir.

4.2.2 Java-program som tolker modell i sanntid

Variant nummer to vil være å tolke modellen på sparket. Dette vil også være den form for modellering som er absolutt enklest for en sluttbruker, da ingen Java-programmering trengs. En spennende konsekvens av dette er at vi kan greit lage interaktive systemer, dersom vi implementerer en sanntids-modus.

XML-modell

Vi kan tenke oss at en bruker vil koble opp sin modell i en generell XML-editor, hvorpå vårt program parser XML-koden til en modell den kan simulere.



Figur 4.4: Flyten i et program som tolker XML-fil til en kjørbar struktur og kjører denne.

Fordeler

Dette systemet fordrer ingen kunnskap til programmeringspråk. Det kan lett utvides til å visualisere systemene, da hver komponent kan ha sin visualiseringsfunksjon. Vi kan dessuten tenke oss at vi vil endre parametre i systemet uten å rekompilere modellen - gjerne rett fra modelleringsverktøyet.

Ulemper

For at sanntidstolkning av en modell skal kunne virke må vi nok sette begrensninger på generaliteten i systemet. Dessuten blir systemet mest sannsynlig en del tregere enn en hardkodet bror som i 4.2.1.

Ut fra argumentasjonen over vil vi å la systemet vårt tolke modellen i sanntid.

Del II

Vår implementasjon av diskret hendelsessimulering i java

I del II vil vi knytte trådene fra diskusjonene i de foregående kapitler og vise vår implementasjon av et system for diskret hendelsessimulering med vektlegging på å studere modellen grafisk.

Først vil vi presentere de byggeklossene vi vil benytte for modellering, før vi ser litt nærmere på implementasjonstekniske detaljer i systemet. Vi presenterer så det grafiske systemet vi har utviklet for å skape modeller til mjmod, før vi avslutter med å se på hvilke muligheter vi har til å studere modellenes korrekthet bla. ved hjelp av det dette.

"Computers can only do what they are told to do."
-Lady Ada Lovelace i sine memoirer

Kapittel 5

Byggeklossene

Vi nevnte i 1.2 at vi nok ikke vil klare å lage et språk som kan modellere alle typer systemoppførsler. Dette gjelder i enda sterkere grad for modellering i grafiske grensesnitt, da vi vil dekonstruere modellene til ikoner og buer og dermed nødvendigvis legger enda strengere begrensninger på oss. Derfor har vi valgt å fokusere på købaserte modeller. I slike systemer vil alle hendelsene i systemet involvere enhetenes bevegelser - hvordan de ankommer og forlater køer. Dermed kan vi konsentrere oss om å modellere nettopp deres vandring gjennom eller i systemet.

Modellene vi ønsker å simulere tenker vi oss at kan bygges opp av byggeklosser, eller som Zeigler kaller dem; *basismodellene*[35] bundet sammen av kanter.

De byggeklossene, som vi nå har implementert er basert på forskning gjort av Odhabi, Paul og Macredie[22, 21] og deres H-ACD-symbolsett (figur 2.4) for firefasetilnærmingen. Vi har innført noen ikoner H-ACD ikke har, samt tillagt egenskaper til enkeltikoner som H-ACD har spredt på flere ikoner.

Figur 5.1 viser ikonene vi har utviklet til mjmod.[1] Disse benytter vi både til grafer av modeller, samt i grafiske grensesnitt for modellering.

Navngivingen på byggeklosser og deres klasser er (stort sett) hentet fra artikkelen *Object-Oriented Simulation* av Joines og Roberts fra [5] (side 397-427).

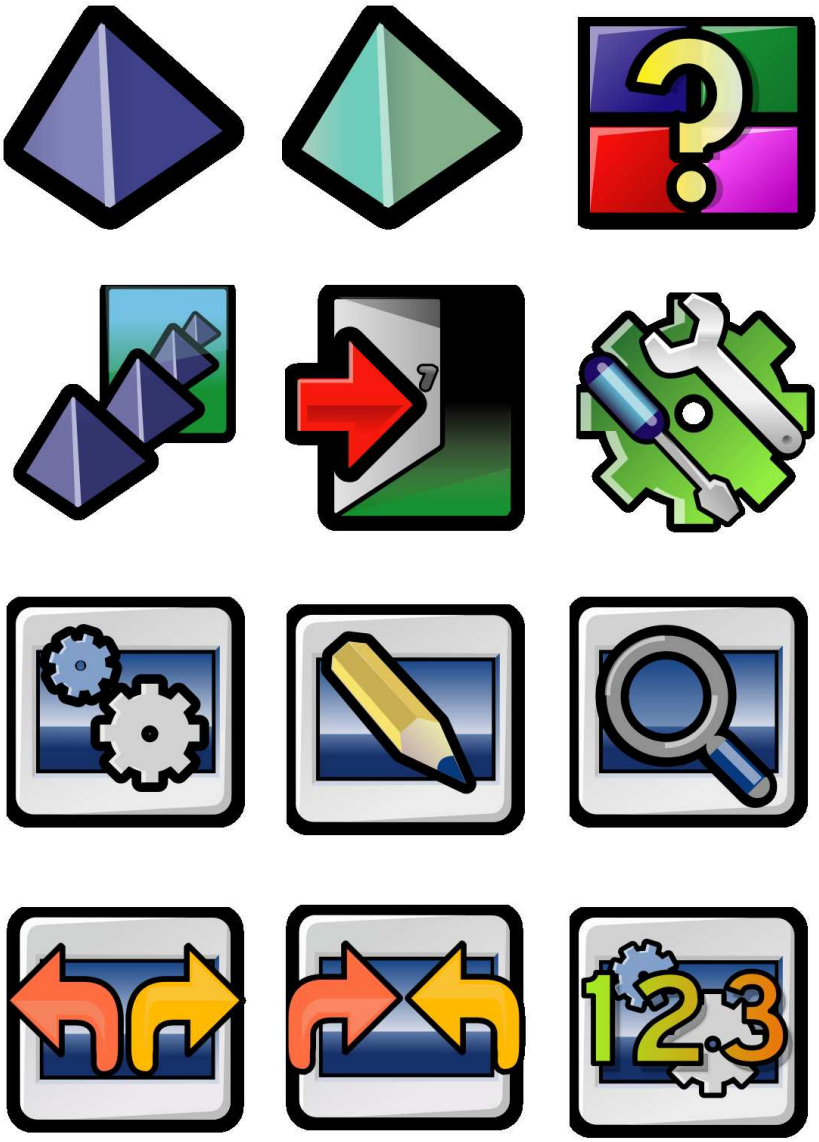
I dette kapittelet vil vi presentere byggeklossene våre nærmere.

5.1 Enheter

Enheter blir ført gjennom modellen og er passive "kunder" av systemet. De brukes til å samle statistikk. Enhetene som representerer bilene i en bilvask vil være typiske eksempler. Enheter kan også representere redskaper til ulike prosesser, som vaskemennene i samme system.

Vi skiller mellom *statiske* enheter som er i systemet hele tiden (eller kan sees på som *redskaper*), og *dynamiske* enheter som kan gå ut av systemet (eller kan sees på som *kunder*). Dette er inspirert av Simscript sin tilnærming til enheter i et simuleringssystem. (Se 2.6.1 på side 14.) Slik oppnår vi et greiere system modelleringsteknisk sett, samt at vi lar de ulike typer enheter samle ulik statistikk. Ved å gjøre denne vrien får vi dessuten muligheten til å avbryte en

[1] Se tillegg A for mer informasjon om redskaper brukt under utformingen av ikonene.



Fra øverst til venstre har vi: *Dynamisk enhet, Statisk enhet, Fordeling, Kø, Utslagsvask, Generator, Ressurs, Taggskriver, Taggleser, Splitter, Fletter og Prioritets-ressurs.*

Figur 5.1: Ikonene vi har laget for å representere byggeklossene i mjml.

simulering selv om aktiviteter har betingelsene sine oppfylt. Dette oppnås ved å se når systemet er tomt for *dynamiske* enheter.

Enheter kan dessuten brukes som brytere eller meldinger som kan flyte i systemet for å fortelle ulike aktiviteter når de kan starte.

5.1.1 Dynamiske enheter

Dynamiske enheter samler statistikk om hvilke køer de har befunnet seg i og hvilke "prosesser" de har inngått i. De rapporterer hvor mye tid de har brukt i kø og i aktivitet til utslagsvasken de ender i.

5.1.2 Statiske enheter

Statiske enheter vil oppholde seg i systemet fra simulasjonens start til slutt, og samler statistikker om feks. hvor lenge en enhet har inngått i en aktivitet - vært aktive - og hvor lenge den har stått i kø - vært passiv, samt antall ganger den har vært aktiv.

5.2 Fordelinger

Java har som nevnt i det medfølgende biblioteket funksjoner for å arbeide med pseudotilfeldige tall (*java.util.Random*), men for å generere slike fra forskjellige fordelinger som passer til diskret hendelsessimulering har vi implementert et eget grensesnitt oppå dette. Dette er basert på Helsgaun sin pakke for å generere tilfeldige tall fra javasimulation.

Fordelingsobjektene (*Distribution*) festes til ulike aktiviteter og prosesser for å gi dem tilfeldige verdier i tid og/eller andre attributter.

Vi har følgende typer fordelinger til rådighet:

- Uniform fordeling (*uniform*)
- Normalfordeling (*normal*)
- Negativ eksponentialfordeling (*negexp*)
- Poissonfordeling (*poisson*)
- Konstant (*constant*)

Felles for alle disse er at de har en en parameter middelverdi (*mean*) som må spesifiseres. I tillegg har vi følgende spesialtilfeller av fordelinger:

- Heltall uniformt mellom to tall (*integer*). Denne krever at vi setter to ekstra-parametre; minimumsverdi og maksimumsverdi på heltallene den skal returnere.[2]
- Les fra fil (*file*).

[2] En liten implementasjonsteknisk detalj er at siden fordelingene skal være så generiske som mulig returnere alle verdier på formen *double* - selv fordelingen *Integer* som man gjerne skulle forvente returnerte *int*. Siden *int* er en delmengde av *double* taper vi ikke presisjon ved dette.

5.3 Køer

Køer utgjør en svært sentral del av vårt simulasjonssystem. De definerer betingelsene til aktivitetene. Alle objekter som skal plasseres i en kø må være klargjort for det, altså implementere "kan settes i kø" (*Queable*). Vi har følgende type køer:

- FIFU-kø (*FIFO-queue*), "først inn, først ut"-kø: Dette er den mest brukte køen. Enheter som ankommer køen først forlater den også først.
- SIFU-kø (*LIFO-queue*), "sist inn først ut"-kø: Dette kalles ofte for en stabel (*stack*). Overfylte heisvogner er eksempel på bruk av slike.
- Prioritets-kø: Her plukker vi ut enheter fra køen i en prioritert rekkefølge. Vi kan tenke oss ulike prioriteringer utfra attributter på enheten som feks. størrelse, ansiennitet eller tid siden ankomst i systemet (i motsetning til køen) gjerne spesifisert ved en tagg. Om vi modellerer et sykehus kan vi tenke oss pasient-enheter sortert etter hvor kritisk tilstanden dens er.

Alle køer kan defineres med en størrelsesgrense, dersom modellen vår trenger det. Andre tenkelige begrensninger på køer defineres av andre byggeklosser. For eksempel kan vi tenke oss at enheter har en gitt tid før de gir opp å vente i køen. Dette spesifiseres i enheten, men det er køen som står for å fjerne utgåtte enheter. Vi kan så tenke oss at køen spør enheter om de er *utgåtte* og i så fall fjerner dem.

I eksempelet med bilvasken kan vi se på køen med biler som venter på å bli vasket og pauserommet som køer.

5.4 Prosesser

Prosesser er ikke avhengig av eksterne tilstander, bare av klokken og sin egen fordeling for iverksettelse. Alle prosesser har en peker til et tidsfordelingsobjekt. Dette objektet sier hvor ofte prosessen skal aktiveres.

5.4.1 Generator

En generatorer er et eksempel på en prosess i mjmod.

Generatoren har altså følgende attributter:

1. En peker til en et tidsfordelingsobjekt. Dette objektet sier hvor ofte prosessen skal aktiveres.
2. En peker til et "mønster"-objekt, som må være av typen *Entity* og implementere *Clonable*.
3. En peker til en kø for levering av instanser av "mønster"-objektene.
4. Et heltall som sier om generatoren skal skape et begrenset eller ubegrenset antall "mønster"-objekter og eventuelt hvor mange.

Denne initieres altså med et mønster på objektene (enhetene) den skal generere. Den leverer disse i det tempoet fordelings-objektet knyttet til den sier og så

mange som den har fått beskjed om i spesifikasjonen av modellen. Generatorer kan sees på som inngangsporten til systemet man vil modellere.

For eksempel vil innkjørselen til bilvasken kunne sees på som en generator. Der den generer kunder til bilkøen i bilvaskesystemet ut fra fordelingen den har koblet til seg.

5.5 Aktiviteter

En betinget hendelse kaller vi for en aktivitet. Når en aktivitet har fått oppfylt sine betingelser for oppstart vil den bli sett på som en prosess den også. Dette løses enkelt i java ved at *Activity* er implemtert som en subklasse av *Proc*, og dermed kan sees på som det inntil den har utført det den skal og er klar for å bli sett på som en *Activity* igjen.

5.5.1 Ressurs

Ressurser (*Resource*[3]) er aktiviteter som lytter på vilkårlig mange køer og startes når alle disse har elementer. Da tar ressursen enhetene fra køene sine og holder dem så lenge som fordelingen knyttet til ressursen sier, før den slepper enhetene ut i utkøene sine. Selve vaskehallen i bilvasken kan bli sett på som en slik ressurs.

Enheter som blir plukket opp i en inn-kø, blir plassert i den respektive ut-køen. Altså fra inn-kø nummer 1 til ut-kø nummer 1, fra inn-kø nummer 2 til ut-kø nummer 2 osv. Dette gjør det enkelt å få oversikt under konstruksjonen av den logiske strukturen til modellen.

Vi kan også tenke oss at en aktivitet krever flere enheter fra samme kø. Dette løses ved å peke til samme køen som flere inn-køer.[4] Dette kan benyttes til for eksempel å simulere en vareheis som bare starter dersom den er full.

I bilvasken kan vaskemaskinen sees på som en ressurs. Den starter hvis og bare hvis det er enheter i køen med biler som skal vaskes og ledige vaskemenn på pauserommet.

5.5.2 Prioritetsressurs

Prioritetsressurser (*Priority resource*) trenger bare enheter på en av innkøene. Den henter enheter prioritert etter rekkefølgen på inn-køene. Enheter som ankommer inn-kø 1 har høyere prioritet enn enheter i inn-kø 2, enheter i inn-kø 2 har høyere prioritet enn dem i inn-kø 3 osv. Prioritetskøen plukker alltid fra den høyest prioriterte køen med enheter i.

Denne type aktivitet kan brukes til å modellere for eksempel forkjørsveier i et veinettverk og QoS i et datanettverk. I bilvasken kan vi tenke oss at vi innførte en VIP-kø av biler som alltid skulle bli vasket før andre biler og dermed implementerte det som en prioritetsressurs for å modellere dette.

[3]Vi gjør oppmerksom på at Paul et al[21] bruker termen resource på en annen måte.

[4]En liten notis om måten vi reint implemtasjonsmessig har løst dette: Ved å innføre en boolsk verdi "se på" som settes på de enhetene som er aktuell mens man sjekker betingelsene til en aktivitet. Dersom en enhet allerede er "sett på " vet vi at denne ikke er tilgjengelig fra denne køen i denne omgang. "Se på" blir nullstilt etter hver aktivitets er ferdig med dens undersøkelse av betingelsene. Dette fører til noen ekstre operasjoner i hvert kall til *contiotions()* men det er en liten pris for hva dette gir modelleringsteknisk.

5.5.3 Fletter

En fletter (*Merger*) minner mye om en ressurs men tar elementer fra en eller flere køer og skaper et nytt objekt av en gitt type.

Dessuten kan en fletter lytte etter når en eller flere køer går tom og fylle dem med enheter av en gitt type.

En begrenset såpebeholder i bilvasken kan modelleres ved en ekstra innkø-betingelse til vaskemaskin-ressursen med såpeelementer, samt en fletter som fyller på såpe når det går tomt.

5.5.4 Taggskriver

En taggskriver (*TagWriter*) fester tagger til enhetene. Verdiene til disse taggene henter den fra en tilkoblet fordelingsobjekt, navnet på taggen spesifiseres under initiering av skriveren.

Taggskriveren henter enheter fra en kø og plasserer dem i en annen, med en pause gitt fra et annet fordelingsobjekt.

Taggskrivere kan benyttes til å representere et "ønske" eller en "plan" til en enhet. I bilvasken kunne vi for eksempel modellert hver bils valg av vaskeprogram ved å gi dem tagger. Deretter kunne vi benyttet tagglesere og splittere for å hente dem ut under bilenes drift gjennom bilvasken.

5.5.5 Splitter

En splitter (*Splitter*) tar enheter fra én kø og plasserer dem i én av vilkårlig mange ut-køer ut fra en tilkoblet fordeling og frekvenser oppgitt under spesifiseringen av en splitter. Den kan også splitte en kø ut fra gitte tagger på enhetene i den. Da må verdien på taggen den leter etter være et heltall for å vite hvilken ut-kø den skal i.

Om vi innfører en kvalitetsikring ved endt bilvask kan dette implementeres som en splitter. I stokastiske modeller kan man spesifisere at én av 20 biler er missfornøyd og må vaskes om igjen.

5.5.6 Taggleser

En taggleser (*TagReader*) leter gjennom en kø til den finner en enhet med taggen den leter etter, hvorpå den venter en tid spesifisert av fordelingsobjektet sitt og plasserer enheten i utkøen sin. En taggleser kan dessuten ha ekstrabetingelser som skal være oppfylt ved å knytte andre innkøer og eventuelt utkøer til den.

En eksempel på modellering ved hjelp av en taggleser i forbindelse med bilvasken kan være at biler skal ha ulike vaskeprogrammer.

5.6 Utslagsvask

I utslagsvasken (*Sink*) ender objektene som er på vei ut av systemet. Denne arver egenskaper både fra aktiviteter og køer - et eksempel på en aktivitet med en intern kø.

Den sanker staistikk fra enhetene den får. Dette gjør den kontinuerlig under kjøring av simuleringen slik at vi kan frigjøre minnet disse nå redundante en-

hetene har brukt, ved å gi dem til søppelsamleren (*Garbage Collector*) til Java. Dette er inspirert av implementasjonen til Helsgaun[13].

På samme måte som generatoren kan sees på som en grense til systemet man vil modellere, kan utslagsvasken også det, nemlig utgangen fra det.

Til utslagsvasken er det også knyttet en log-fil, der data om enhetene den mottar blir skrevet. Denne filen kan så etterprosesseres for å studere resultatene av simulasjonskjøringen dypere.

For å knytte også denne byggeklossen til bilvasken kan vi presisere at utkjørselen på bilvasken typisk vil modelleres som en utkjørsel i mjmod.

5.7 Spesifisering av modellen

For å spesifisere og ta vare på en modell, har vi utviklet et XML-filformat kalt *My Java Modelling Language,* eller mjml. Den vanligste måten å beskrive en slik XML-basert filtype er ved hjelp av en såkalt DTD - dokumenttype definisjon *(Documnet Type Definition)*[5]. Hensikten med denne er å definere de gyldige byggeblokkene i et XML-dokument. Den definerer dokumentstrukturen med en liste over gyldige elementer.[6] I Tillegg D er det eksempler på mjml-filer samt en fullstendig DTD for mjml. Vi vil i dette kapittelet derimot gå mer i detalj på spesifiseringen av en modell i mjml.

5.7.1 Filstrukur

Alle elementene som inngår i systemet spesifiseres altså ut fra XML-biter ala det vi ser i figur 5.2. Her ser man at koden til selve vaskemaskinen i bilvasken. Først en tagg som sier hvilket nummer objektet har, så et felt med navnet. Det hadde strengt tatt holdt med det ene av dem, og da helst navnet, men pga kluss med tegnkoding i filsystemer, besluttet vi at det er tryggest å bare holde seg til nummer-referanser for oppkobling av modellen.

[5]DTD ble i sin tid utviklet for å beskrive SGML, som XML er en delmengde av.

[6]Bemerkning: DTD'er kan minne i oppbygging om Backus-Neur-notasjon (BNF), som er en kontekst-fri gramatikk for definering av gyldig syntaks i formale programmeringsspråk. Opprinnelig laget av Backus for Algol 60.

```
<activity>
        <no>1</no>
        <name>Vaskemaskin</name>
        <type>Resource</type>
        <distribution>Vasketid</distribution>
        <distribution-pointer>2</distribution-pointer>
        <inqueues>
                <queue>
                        <name>Bilkø</name>
                        <pointer>1</pointer>
                </queue>
                <queue>
                        <name>Pauserom</name>
                        <pointer>2</point>
                </queue>
        </inqueues>
        <outqueues>
                <queue>
                        <name>Utkjørsel</name>
                        <pointer>3</pointer>
                </queue>
                <queue>
                        <name>Pauserom</name>
                        <pointer>2</pointer>
                </queue>
        </outqueues>
</activity>
```

Figur 5.2: Definisjon av en aktivitet i bilvasken

Den neste taggen går da på hvilken type aktivitet dette dreier seg om - i dette tilfellet en ressurs. Så kobler man på en fordeling på ressursen som forteller hvor lang tid den skal bruke på å gjøre det den skal gjøre, i dette tilfellet vaske biler. Her medfører filsystem-problematikken nevnt over at vi må operere med redundant informasjon - både navn og peker til fordelingen. Så kommer vi til spesifiseringen av selve køene ressursen skal hente enheter fra og siden plassere dem på. For eksempel på en fullstendig mjml-kode se D.4 på side 110.

Legg merke til at alle forbindelser mellom komponenter er strengt definert ved hjelp av innporter og utporter. Formatet korresponderer dermed med en graf som kan settes opp i en GUI. Grunnet behovet for ulik prioriteringer på aktiviteter definert av rekkefølgen på dem, som jeg kommer tilbake til siden, kan vi ikke spesifisere *alt* reint grafisk. Ekstra argumenter kan evt. oppgis i oppsprettvinduer.

Kapittel 6

Simulasjonssystemet utdypet

Simulasjonssystemet vi har implementert har vi valgt å kalle mjmod (*my java modelling*).

6.1 Utdypning av simulatoren

Vi har laget en klasse *Simulation* som er selve simulatoren. Den holder styr på tidskøen og listen av aktiviteter.

Dermed fungerer den som *simulasjonslederen* - selve hjertet av simuleringen. Denne funksjonaliteten er implementert som funksjonen *simStep()* ut fra fasene vi spesifiserte i 2.5.4 på side 12. (Kildekode i tillegg C.1.)

Vi har implementert simulatoren etter mønsteret fra avsnitt 2.5.4 på side 12; en modifisert trefasetilnærming.

6.1.1 De tre fasene

Legg merke til at i tidskøen ved simulasjonsstart vil alle *ubetingede* aktiviteter - det vi har kalt for *prosesser* - allerede være.

I fase 1 spør den aktivitetene en og en om noen av dem har betingelsene sine oppfylt. Måten dette er løst på i praksis er at vi har en lenket liste som inneholder alle aktivitetene som er spesifisert i modellen. Vi gjør et enkelt kall til en rutine kalt *conditions()*[1] i hvert element i listen. Aktiviteter som har betingelsene sine oppfylt regner dermed ut hvor lang tid aktiviteten de skal utføre tar og setter sin tid for fullførelse internt og returnerer *sann*. Dette er et tegn til simulasjonslederen at aktiviteten nå *må* skje og at de skal legges i tidskøen som en prosess. Samtidig som en aktivitet gjør dette tar den til seg de enhetene som gjorde at betingelsene dens var oppfylt og plasserer dem i sitt interne lager. Nå inngår disse i aktiviteten det er snakk om. Dette gjør at ingen andre aktiviteter kan "bruke" de samme enhetene for å få tilfredstilt sine betingelser.

Etter å ha interrogert alle aktiviteter i systemet sjekker vi om tidskøen er tom. Er den det er simulasjonen over. Da er det ingen flere hendelser som venter, altså ingen flere endringer i systemets tilstand, og dermed ingen flere aktiviteter

[1] *conditions()* er definert for alle aktiviteter da den er spesifisert i den abstrakte klassen *Activity* som alle aktiviteter implementerer.

som kan få betingelsene sine oppfylt. Er det elementer i tidskøen tar vi fatt i det første av dem og setter systemtiden til tidspunktet det er satt til å starte. Dette er det vi kaller fase 2.

Fase 3 tar for seg kjøring av elementet som er først i tidskøen. Om vi ikke har nådd maksimal simulasjonstid kjøres *actions()* i dette elementet vi hentet ut fra tidskøen. Dermed gjør aktiviteten eller prosessen det er programmert til å gjøre for så å returnere en verdi som sier om den fortsatt har betingelsene sine oppfylt. Dette gjelder alltid for prosesser, men vi kan se for oss systemer der dette også gjelder for enkelte noen aktiviteter.

Om vi derimot har nådd makstiden[2] skal vi la elementet utføre sine *actions()* hvis og bare hvis det er en aktivitet.

Deretter tar vi en ny iterasjon.

6.1.2 Et generelt rammeverk

Det resulterende systemet er et generelt rammeverk for simulering implementert ved aktivitetsinterrogering. *Simulation*-klassen er, som vi var inne på, sentral i alle simuleringer. Måten vi skaper en kjørbar simulering er at vi lar klassen for simulering vi vil skape være en utvidelse av klassen *Simulation*. På den måten arver den den funksjonaliteten den trenger for å kjøre simuleringer. For å programmere modeller i dette kan man implementere aktiviteter basert på klassen *Activity*. Disse må man gi en rutine kalt *conditions()* og en rutine kalt *actions()*. Likeledes kan man implementere prosesser basert på *Proc*. Disse trenger bare å spesifisere *actions()*. Videre kan en bruke *Distribution* for stokastiske verdier, og kø-klassene for køer.

6.2 Utvikling av *modelParserGUI*

Det vi har så gjort er å utvikle en tolker som tar en mjml-fil spesifisert i foregående kapittel og setter opp en kjørbar struktur til vår simulator. Denne tolkeren har vi valgt å kalle *modelParserGUI()* og er en utvidelse av klassen *Simulation()*.

6.2.1 Tolking av XML-fil

Når man nå kjører *modellParserGUI* med en fil i mjml-formatet som inndata, tolkes denne mjml-strukturen og det bygges opp en modellen klar til kjøring.

Dette gjøres ved å instansere de objektene filen spesifiserer samt gjøre nødvendige koblinger objektene mellom. Den starter med å sette opp de ulike fordelingene, enhetene og køene som blir spesifisert i mjml-filen. Disse er uavhengige enheter og kan skapes uten å gjøre noen koblinger. Deretter tolkes prosesser og resurser. Disse får koblet til seg det de trenger for å kjøre. Generatorer trenger køer å putte objektene de kommer til å generere i. De trenger dessuten et fordelings-objekt som kan fortelle dem hvor lang tid som går mellom hvert objekt som genereres, og ikke minst trenger de å vite hvilke objekter de skal lage.

[2] Her har vi muligheten til å sette noen ekstrabetingelser for avbrudd på simuleringen, men det vil vi se på siden.

Aktivitetene plasseres så i en liste som holder oversikt over aktiviteter i systemet, og generatorer og andre prosesser plasseres i listen over ubetingede hendelser, eller prosesser om man vil - altså tidskøen i systemet.

6.3 Resultatshåndtering

Der spesifisering av en simulering reint grafisk går greit i vårt programsystem, er det vanskeligere å spesifisere hvordan vi skal behandle statistiske data som genereres under kjøring. Diskret hendelsessimulering har rett og slett for mange muligheter for datasamling til at det skal la seg gjøres grafisk i en fei, og er derfor utenfor omfanget til denne oppgaven. All statistikk sendes til filer (eller terminal) for postprosessering av bruker.

6.3.1 Oppsamling av statistiske data

Alle objektene samler statistikk som en kan velge å etterprosessere, men spesialredskaper som "utslagsvasker" (*Sink*) kan samle statistikk under kjøring og "kvitte seg med" gamle utgåtte enheter.

Siden for eksempel alle køer ligger i en liste av køer, vil man kjapt hente ut statistikk angående køer ved å gjøre oppslag i denne listen. Til alle objekter i systemet kan man feste en logfil som dette bruker til å skrive til under hele kjøringen av simulasjonen. Disse filene kan etterbehandles i hvilken som helst databehandingsprogram, da de bygges opp som kommadelte tekstfiler.

6.4 Klassiske problem i forbindelse med diskret hendelsessimulering

Det er spesielt to problemstillinger ethvert diskrethendelsessimuleringssystem må ta hensyn til; samtidighet og faren for vranglås.

6.4.1 Samtidighet

Hendelser som er samtidige er og blir et problem i diskret hendelsessimulering. Det ligger i naturen til denne måten å angripe simulering på at ingenting skal hende samtidig, da ting som hender på samme tidspunkt i simuleringen er innbyrdes prioritert. Dette går stort sett bra, men, for å bruke vårt system som eksempel, dersom to aktiviteter lytter på samme kø etter enheter, vil alltid en av dem være prioritert. En av dem vil trekke det lengste strået og rive til seg enheten og dermed må den andre vente på neste enhet. I systemet vårt har vi løst det ved å la de aktivitetene som er definert først i XML-filen som spesifiserer modellen være de første som spørres om deres betingelser er oppfylt. På den måten kan man stokke om på dem til modellen er slik man ønsker.

Dette er en sentral problemstilling i diskret hendelsessimulering som vi kunne ha sett mer på i forbindelse med systemet vårt, men det er utenfor omfanget til denne oppgaven.

6.4.2 Vranglås

Vranglås (*deadlock*) er et fenomen som kan oppstå i en simulering. Dette er en situasjon der vi ikke kommer noen vei i simuleringen, da to eller flere prosesser er innbyrdes avhengig av hverandres fullbyrdelse.

Vikepliktskryss der alle veier inn har vikeplikt for en vei er et typisk eksempel fra den fysiske verden på når det kan oppstå vranglåser.[3] Dersom det kommer biler samtidig på alle armene inn til et slik kryss og alle er like høflige vil vi kunne tenke oss at krysset stopper opp. Da vil en gitt bil vike for den bilen den er regelbundet å vike for som igjen viker for en bil den må vike for osv til sirkelen er sluttet. I naturen løser vi dette ved at noen er mer påståelig enn andre og kjører først, men i simuleringer vil slike ting kunne skape trøbbel og forstoppelse.

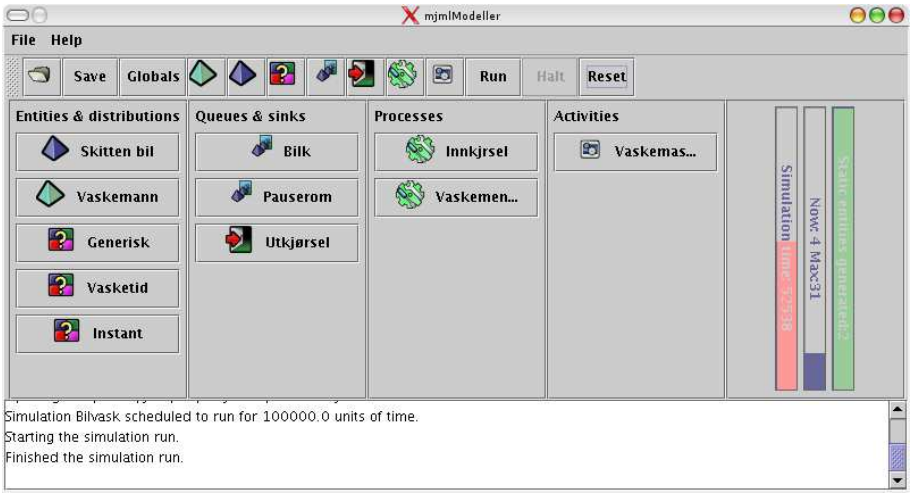
Vi løser vranglåsproblemet på en tilsvarende måte som veikrysset løses i virkeligheten; vi gir alle aktiviteter unik prioritet.

[3] Trådløse nettverk er et annet eksempel der tilsvarende kan hende. Der sender nemlig alle maskinene i et nett på samme frekvens. Dersom alle hadde sendt samtidig ville ingen kunne nådd frem, da alt bare ville blitt kakafoni. Derfor opererer man med prinsippet om dobbelt høflighet. Det vil si at om man vil sende en informasjonspakke sjekker man først om kysten er klar - om ingen andre sender akkurat da. Om dette er tilfelle venter man en tilfeldig liten periode til og sjekker igjen. Om fortsatt ingen sender begynner man sendingen sin. Grunnen til dette tilfeldige oppholdet er nettopp for å unngå vranglås. Hadde dette oppholdet vært en gitt lengde ville sjansen for å snakke i munnen på hverandre vært stor. Dette blir på en måte det motsatte av vranglås, men utfallet er det samme.

Kapittel 7

Grafisk grensesnitt for modellering

For å bygge modeller grafisk har vi utviklet et grafisk grensesnitt vi har valgt å kalle mjmlModeller (*my java modelling language modeller*). Grensesnittet er bygget opp av en redskapslinje, en menylinje, fire kolonner for modellering, en kolonne for sanntidstilbakemeldinger samt et tekstfelt for tilbakemeldinger fra systemet. (Se figur 7.1.) De fire kolonnene for modellering inneholder knapper for å endre parametre til objektene som inngår i modeller.



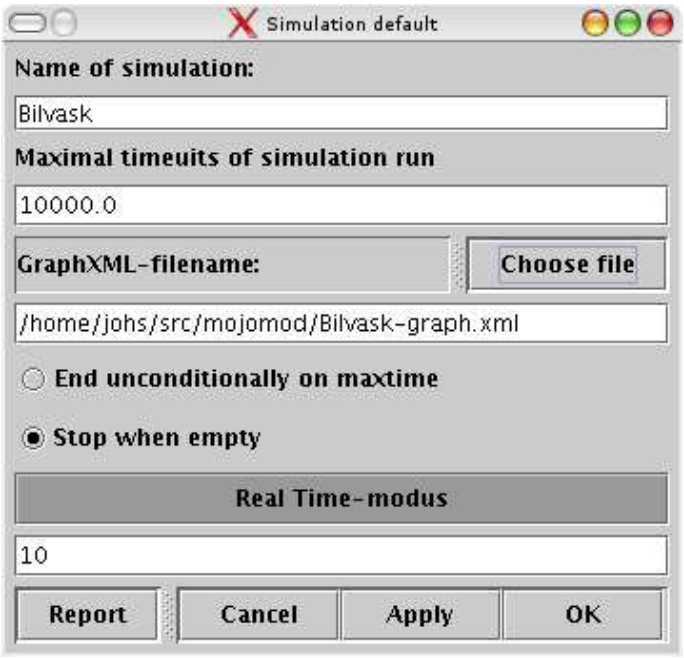
Figur 7.1: mjmlModeller stoppet under kjøring av en bilvaskmodell.

Bruker kan sette opp modeller fra bunnen av i mjmlModeller. Det har vi implementert enkelt og greit ved å manipulere listene simulatoren har over enheter, fordelinger, køer, prosesser og aktiviteter. Slik har man hele tiden mulighet til å teste modellen mens man modellerer den. Mer om dette i 8.1.1, nå vil vi vise hvordan en bruker kan bygge opp en modell.

7.1 Globale innstillinger

Knappen *"Globals"* fører til et nytt vindu som vist i figur 7.2. Her kan bruker sette de globale parameterene til modellen. Knappene og feltene skulle være selvforlarende nok, men legg spesielt merke til knappen "*Real Time-modus*". Den lar en studere simuleringen i "sanntid" (mer om dette i neste kapittel). Tallet i feltet under spesifiserer antallet simulasjonstidsenheter per sekund brukertid.

Ved dette grensesnittet kan vi når som helst gjøre endringer i modellene våre.



Figur 7.2: Spesifisering av globale parametre til en modell i mjmlModeller

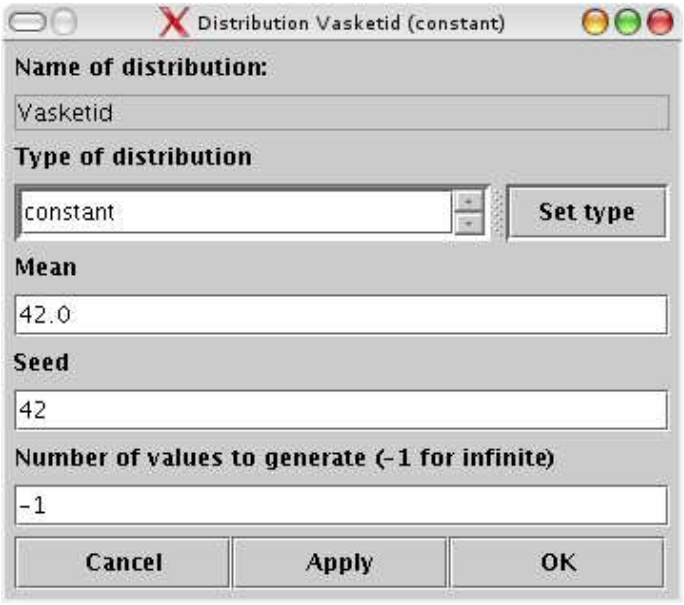
7.2 Enheter

Dynamiske og statiske enheter skapes ved å klikke på deres respektive knapper. Disse spesifiseres bare ved et entydig navn.

Etter å ha godkjent en enhet dukker denne opp i kolonnen "*Entities & Distributions*" i hovedvinduet.

7.3 Fordelinger

Fordelinger legges til modellen ved å klikke "Ny fordeling"-knappen. Da vil bruker få et skjermbilde som vist på figur 7.3. Her kan hun sette de parameterene som en fordeling trenger.



Figur 7.3: Spesifisering av en fordeling i mjmlModeller.

Bruker kan når som helst gå tilbake og endre parametrene til fordelingen ved å klikke på knappen som dukker opp i første kolonne i hovedvinduet etter at hun har akseptert fordelingen. Da får hun tilbake dette redigeringsvinduet. Tilsvarenende gjelder det for køer, prosesser og aktiviteter også.

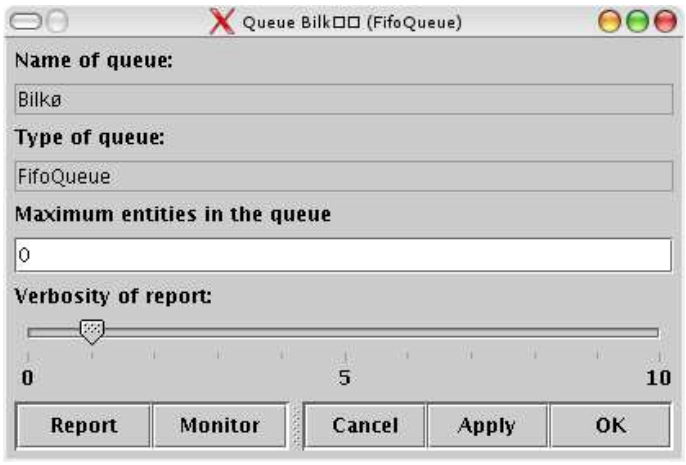
7.4 Køer og utslagsvasker

Ved spesifisering av en kø må bruker først oppgi navn og velge hvilken type kø som skal benyttes i modellen. (Figur 7.4.) Om hun spesifiserer en utslagsvask hopper vi over dette momentet, da vi (foreløpig) bare opererer med én sort slike.

Figur 7.4: Velg kø-type.

Deretter settes de parameterene som definerer køen eller utslagsvasken. Figur 7.5 viser grensesnittet for dette. De to første attributtene, det entydige navnet på fordelingen og typen kø er også ikke mulig å endre på her, men maksimal kø-lengde kan settes og graden av ordrikhet i rapportene kan justeres. (Mer om sistnevnte i seksjon 8.1.1.) Legg merke til at en maksimal kølengde satt til 0

medfører at køen ikke har begrensnig på lengden.[1] Vinduet for å spesifisere utslagsvasken ser litt anderledes ut, da vi der opererer med en logg-fil og kølengden ikke spiller noen rolle.



Figur 7.5: Spesifisering av en kø i mjmlModeller.

Legg merke til de to knappene nederst til venstre i figuren; "Report" og "Monitor". Disse vil vi komme tilbake under behandlingen av sanntidsgrensesnittet.

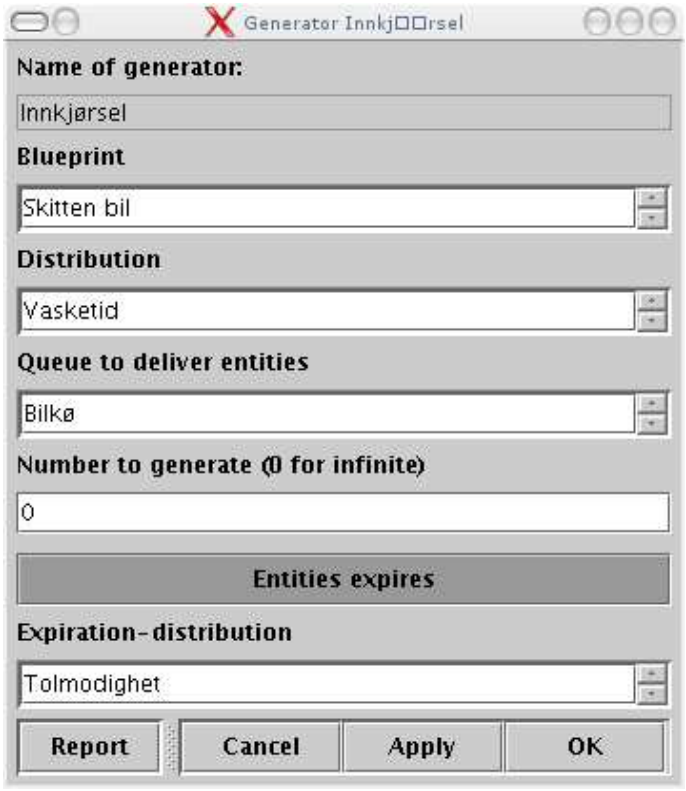
Køene og utslagsvaskene deler en kolonne i mjmlModeller-grensesnittet.

7.5 Prosesser

Generatorene er de eneste elementene som sees på som prosesser i systemet vårt. Disse kan bruker spesifisere i et grensesnitt som på figur 7.6.[2]

[1] Det å velge 0 for å representere uendelig er et velbrukt fenomen i dataverden, men kan være uheldig. Det er ofte bedre å bruke negative tall. I dette tilfellet gir det ikke mening å ha en kølengde på 0, så det går greit. Vi kan tenke oss å lage en utvidelse til køene i mjmod for diverse spesialtilfeller med kølengde satt til 0, men for enkelhetens skyld har vi satt 0 til å representere uendelig.

[2] Vi må ha (minst) en av hver av de foregående objektene før vi kan spesifisere en prosess, siden vi har valgt å koble opp modellene i sanntid.



Figur 7.6: Spesifisering av en generator i mjmlModeller.

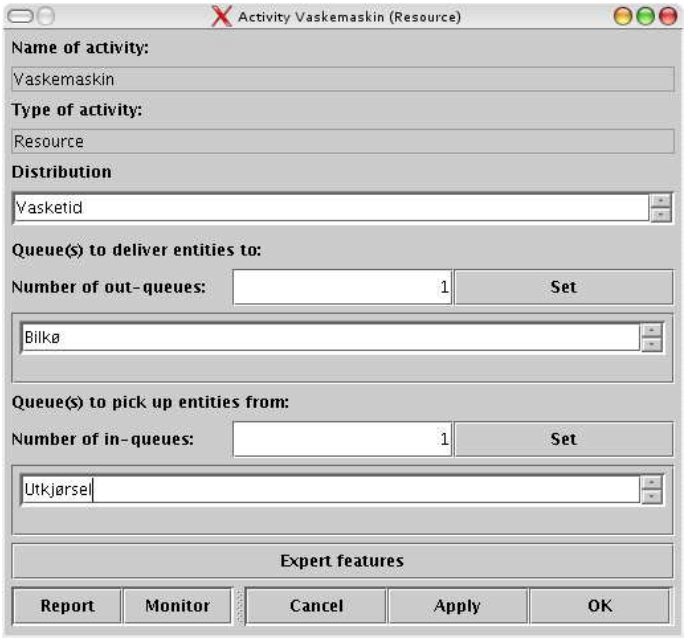
Legg merke til knappen for å markere om enheter utgår. Ved å sette den aktiv kommer feltet for å velge en fordeling for utløpstider for enheter i systemet til syne. Da vil altså enheter produsert av denne generatoren løpe ut på tid gitt av fordelingen for utløpstider.

7.6 Aktiviteter

Når man skal spesifisere aktiviteter må man først velge form for aktivitet i en kombinasjonsboks (Figur 7.7).

Figur 7.7: Valg av form for aktivitet i mjmlModeller.

Deretter får bruker et vindu for å spesifisere den aktuelle aktiviteten. (Se figur 7.8). Noe av det som skiller dette fra vinduet fra forrige avsnitt er listene for å velge innkøer og utkøer. Antall slike samsvarer med feltene "Antall inn-køer" og "Antall ut-køer" og kan settes vilkårlig (større enn 1) når som helst.



Figur 7.8: Spesifisering av en aktivitet i mjmlModeller.

Her har vi skjult noe funksjonalitet under bryteren "Ekspert-egenskaper" (*Expert features*). Dette innebærer valg av hva en aktivitet skal gjøre dersom ut-køen dens er full ol.

De ulike formene for aktiviteter krever ulike parametre, og dermed ser deres grensesnitt anderledes ut, men vi vil ikke gå inn på alle her, da de er i hovedsak lik dette vi har presentert her.

7.7 Lasting av modeller fra fil

"Åpne fil"-knappen åpner en filvelger der man kan søke frem en fil i mjml-format. Ved lasting av en modell fra fil benytter vi tolkeren fra *ModelParser-GUI* spesifisert i seksjon 6.2 på side 42. Denne tolkes altså til struktur som kan kjøres i simulasjonssystemet vårt. Samtidig genereres en GraphXML-fil for visualisering i egnet grafpakke. Etter å ha tolket mjml-filen fyller mjmlModeller knappekolonnene med knapper på tilsvarende måte som ved modellering. Dermed kan bruker redigere modellen på samme måte som om hun hadde laget modellen fra bunnen av, samt legge til nye objekter.

Da er det på tide å kjøre simuleringene våre og få tilbakemeldinger.

Kapittel 8

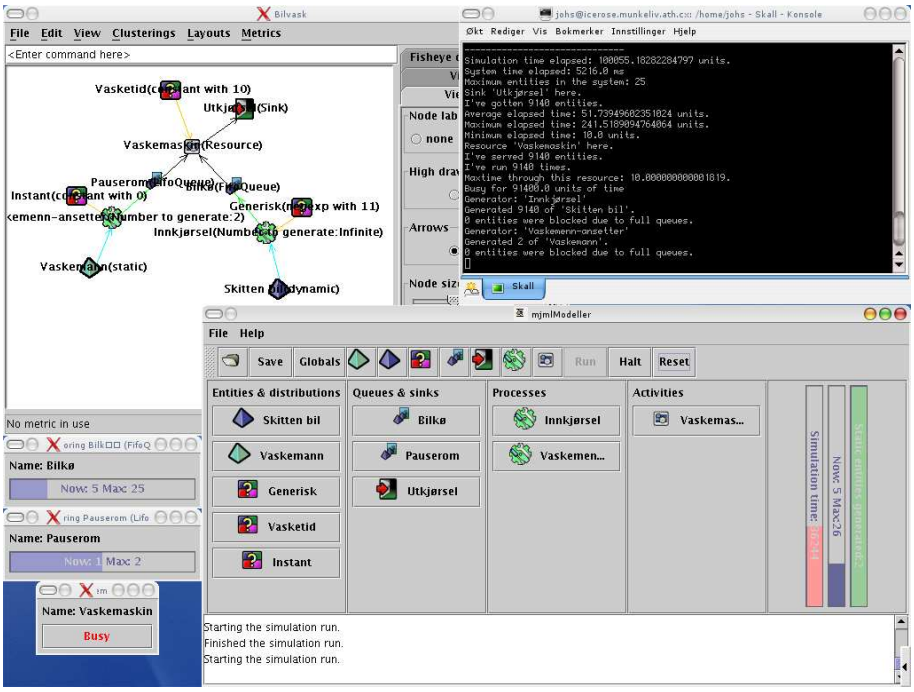
Tilbakemeldinger under og etter simulering

"For a moment, nothing happened. Then, after a second or so, nothing continued to happen."
- Douglas Adams

Tilbakemeldinger fra simuleringssystemet er viktig - både under kjøring og i ettertid. Både av brukervenlighetshensyn[1] og av modelleringstekniske hensyn. Det å kunne observere modellen under kjøring hjelper på oss å oppdage svakheter og mangler i modellene våre.

Dette leder oss til en kritisk fase i modelleringsøyemed; studiet av modellens korrekthet. Kan vi stole på at en gitt modell gir oss en tilnærming til det problemet vi vil løse? I systemet vårt har vi flere måter å finne dette ut på. Figur 8.1 viser et skjermbilde med noen av de redskapene man har under utvikling av en modell i mjml. Øverst til venstre har vi *Royere* som viser logisk struktur i modellen vi ser på. Til venstre for den har vi konsollet som gir tilbakemeldinger fra simuleringssystemet. Under det har vi mjmlModeller der modellen kan manipuleres og simuleringen kontrolleres. Til sist har vi i venstre hjørne tre vindu som overvåker diverse deler av modellen.

[1] Dersom simuleringen har kjørt urovekkende lang tid, er det greit å vite at den ser ut til å komme til å fullføre eller i allefall at noe kommer til å hende.



Figur 8.1: Typisk skrivebord under utvikling av en modell i mjml.

8.1 Grafisk sanntidsgrensesnitt

I mjmlModeller-grensesnittet har vi mulighet til å kjøre modellene vi bygger og/eller laster. (Se figur 7.1.)

8.1.1 Kjøring av modell

Ved å klikke på "Kjør"-knappen (*"Run"*) kjøres modellen. Under kjøring har vi muligheten til å pause simuleringen for å skrive ut rapporter, endre parametre starte overvåking av diverse objekter osv. Vi kan også slå på og av sanntidsmodus og eventuelle monitorer.

8.1.2 Tekstbaserte tilbakemeldinger

Tekstpanelet nederst i vinduet gir tilbakemeldinger relatert til vår interaksjon med det grafiske grensesnittet, mens tilbakemeldinger, som statistikk og lignende, fra selve simulasjonen, går til konsollet vi startet mjmlModeller fra. Dette for å skille meldinger fra det grafiske grensesnittet fra meldinger fra simulasjonssystemet. Meldinger fra simulasjonssystemet kan dessuten dermed kjapt omdirigeres (til tekstfiler) for etterbehandling.

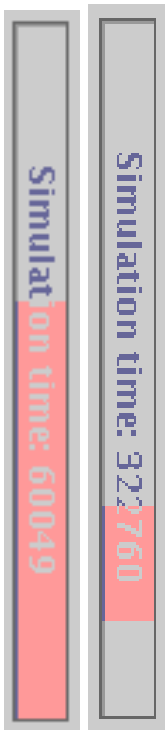
Vi kan når som helst under kjøring av en simulering be om en rapport fra et objekt som inngår i den. Dette gjøres ved å trykke "Rapport"-knappen (*"Report"*) i vinduene der bruker kan sette parametre for de respektive objektene når som

helst under simuleringen; *modelleringsvinduene*. (Se figur 7.2, 7.6,7.5 og 7.8. Disse tilsvarer de rapportene man får ved simulasjonsslutt.

I forbindelse med rapporter fra køer kan bruker spesifisere graden av ordrikhet (*"verbosity"*) i rapporten. Ved lav ordrikhet får bruker bare generelle statistikker om maksimal og eventuelt nåværende kølengde samt gjennomsnittstid og maksimal tid i aktuell kø. Ved en ordrikhet på over 5 vil hun også få skrevet ut stemplingskortene til enhetene i køen. Slik kan vi se hvor disse har vært før de nådde denne køen. Ved å øke ordrikheten til over 7 vil vi i tillegg få skrevet ut eventuelle tagger som er festet på enhetene i køen. Dette kan også være nyttig for å se at modellen oppfører seg som vi har planlagt.

8.1.3 Global overvåking under kjøring

Til høyre i mjmlModeller-vinduet har vi tre fremdriftsstolper. Den første viser simulasjonstiden i systemet. Den går fra 0 til maksimaltiden satt globalt i systemet. Dersom simulasjonen går utover dette er det ikke mulig å vite hvor lang tid dette vil ta og framdrifsstolpen skifter til såkalt ubestemt-modus for å indikere nettopp dette. (Se8.2.)



Figur 8.2: Fremdriftsstolper som viser tiden i en simulering - en bestemt og en ubestemt.

Den midterste stolpen sier hvor mange dynamiske enheter som er i systemet under kjøring i forhold til maksimalt antall enheter i aktuell simulasjonstid. Denne stolpen endrer skala etterhvert som maksimalt antall enheter i systemet

endrer seg, da vi på forhånd ikke vet i hvilken størrelsesskala antall enheter befinner seg i. Stolpen lengst til høyre viser antall statiske enheter generert til nå i simulasjonen.

8.1.4 Overvåking av køer og aktiviteter

I tillegg til disse globale fremdriftsstolpene kan vi velge å overvåke vilkårlig mange køer og aktiviteter som inngår i modellen. Figur 8.3 viser en overvåking av de to køene vaskemaskinen er knyttet til, samt vaskemaskinen selv. Disse overvåkerene kan slås på med knappen "overvåk" (*"Monitor"*) i modelleringsvinduene til objektene. (Se figur 7.5 og 7.8.)

Figur 8.3: Sanntidsovervåking av en vaskemaskin og dens to inn-køer.

Overvåking av køer tilsvarer overvåking av antall dynamiske enheter i systemet. Fremdriftsbjelken viser kølengde i aktuell simuleringstid i forhold til maksimal lengde. Her også vil maksimal lengde hele tiden justeres da det ofte er vanskelig å anslå på forhånd hvor lange køer er.

Aktivitetsovervåkeren viser status for overvåkede aktivitet; rød "Opptatt" eller grønn "Ledig".

8.1.5 Sanntidsfunksjonalitet

I forbindelse med disse overvåkerene, kan vi bestemme at simulasjonstiden skal samsvare på et sett med brukertiden, altså en form for sanntidsfunksjonalitet. Ved å få simulasjonstiden til å flyte jevnt slik er det lettere å følge med på hvordan enheter beveger seg gjennom systemet.[2] Denne kan slås av og på i det globale kontrollpanelet for simuleringen. (Se figur 7.2.) Her kan vi også spesifisere hvor mange simulasjonstidsenheter som maksimalt skal passere per brukertidssekund.

8.2 Visualisering av modellen

For å visualisere den logiske strukturen i modellene vi setter opp i systemet vårt benytter vi GraphXML og en grafbehandler som støtter denne filtypen. Vi har valgt Royere til dette formålet.

[2]Dette løser vi, reint implementasjonsmessig, ved å legge inn en tikker i systemet som passer på at modelltiden hele tiden er en funksjon av brukertiden.

8.2.1 Litt om GraphXML

GraphXML er en foreslått XML-standard for håndtering av grafer.[3] Det er utviklet ved CWI ("Senter for Matematikk og Datavitenskap") i Amsterdam.

Det finnes flere programmer for å visualisere grafer spesifisert i dette formatet, feks.:

- Hypergraph (http://hypergraph.sourceforge.net/)
- Royere (http://gvf.sourceforge.net/)

I GraphXML kan man spesifisere det meste av det vi er ute etter å visualisere i forbindelse med modelleringen vår. I tillegg til standard grafspesifisering støtter for eksempel språket:

- navn på noder.
- forskjellige tegnestiler og farger for noder for å skille ulike typer noder fra hverander. Man kan til og med tilegne noder små grafikkfiler; ikoner.
- data koblet til noder i form av pekere til filer eller internettsider. Dette kan brukes til å feks. koble pekere til logfiler fra de ulike objektene direkte i grafen. Om man da lurer på detaljer feks. vedrørende en utslagsvask er de statistiske data bare et dobbeltklikk unna.
- navn på kanter. Dette bruker vi til å holde styr på innport/utport-rekkefølgen i modellen.

8.2.2 Modellen i Royere

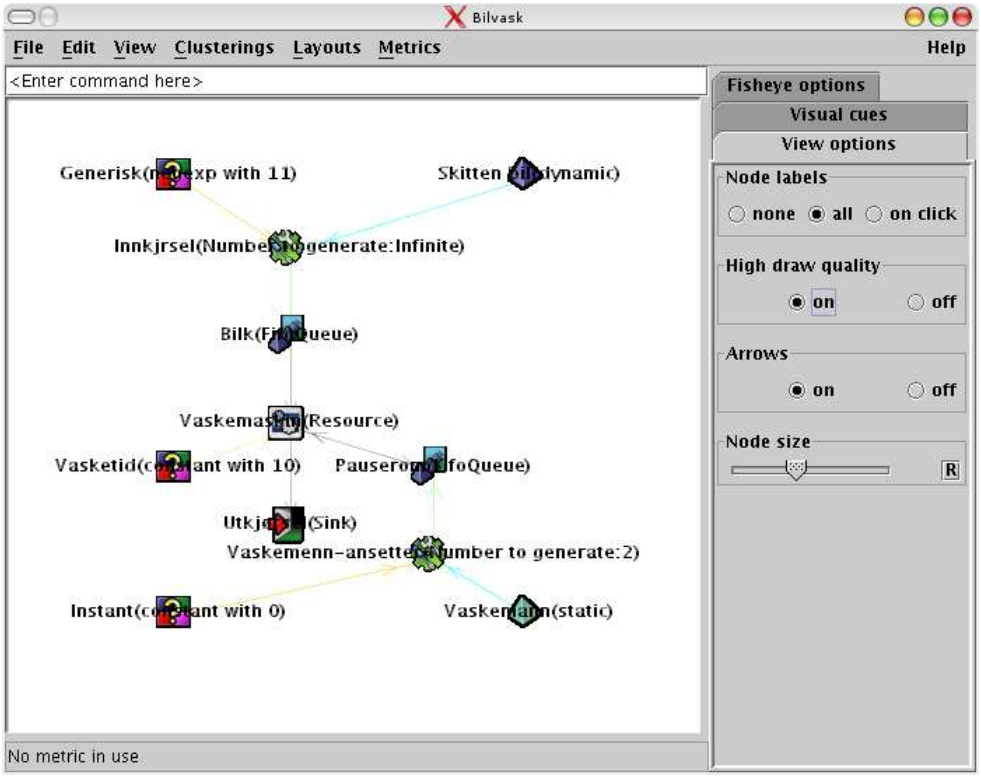
Royere er et grafbehandlingsverktøy utviklet i Java av tidligere nevnte CWI og lansert under GNU/GPL-lisensen.[4] Vi benytter det til visualisering av den logiske strukturen i modellene våre, da dette programmet leser GraphXML og har en del funksjonalitet som vi er ute etter. (Se figur 8.4)

Layout

Det å representere en graf i to dimensjoner på den mest hensiktsmessige måte er en komplisert affære og utenfor omfanget til denne oppgaven, men Royere tar hånd om det for oss.[14] Vi kan velge mellom flere ulike strategier; feks. *Reingold-Tilford, Random 3D Layout, Fruchterman-Reingold, GEM Force-directed og Ring.* Av disse er nok *GEM Force-directed* den mest anvendelige strategien for de fleste av våre grafer. I Royere er den implementert semi-tilfeldig, slik at dersom man etter en kjøring av algoritmen ikke får et oversiktlig nok bilde av grafen, kan man kjøre denne samme om igjen og den vil stokke om på deler og man kan ende opp med en mer oversiktlig graf.

[3] Et eksempel på bruk innen forskning er *Seoul National University* sine studier av avhengeighetsgrafer for leksikalsk informasjon. (http://cogsci.snu.ac.kr/jcs/abstracts%202-1/6%20-%20lea.htm)

[4] GNU/GPL-lisensen (Gnu's Not UNIX - General Public License) er et av Richard Stallman sine viktigste bidrag til datavitenskapen, da denne lisensenformuleringen, ved å sette regler for fri programvare har muliggjort mange av prosjektene basert på åpen kildekode, som for eksempel Linux og relaterte GNU-prosjekter.



Figur 8.4: Bilvask med én vaskemaskin visualisert i Royere.

Forstørrelsesglass og fiskeøyer

I tillegg til vanlig zoom og panorering har Royere også en funksjon for å "zoome" inne på uoversiktlige deler av grafen ved hjelp av et fiskeøye-system. Vi kan definere opp til tre ulike fokuspunkter og justere forvrengningsgraden uavhengig på hvert enkelt av dem. Dermed kan en få oversikt over deler av grafen og la andre deler utheves i mindre grad.

8.2.3 Modellen oversatt til XHTML

En annen måte å få oversikt over mjml-filen på er å oversette den til et annet visuelt XML-format. Ved å definere en overgangsfunksjon i konverteringsspråket *XSL Transformations* (XSLT), kan vi la en XSL-motor oversette fra mjml til, for eksempel, XHTML for å vise strukturer i modellen i en nettleser.

Kort om XSLT

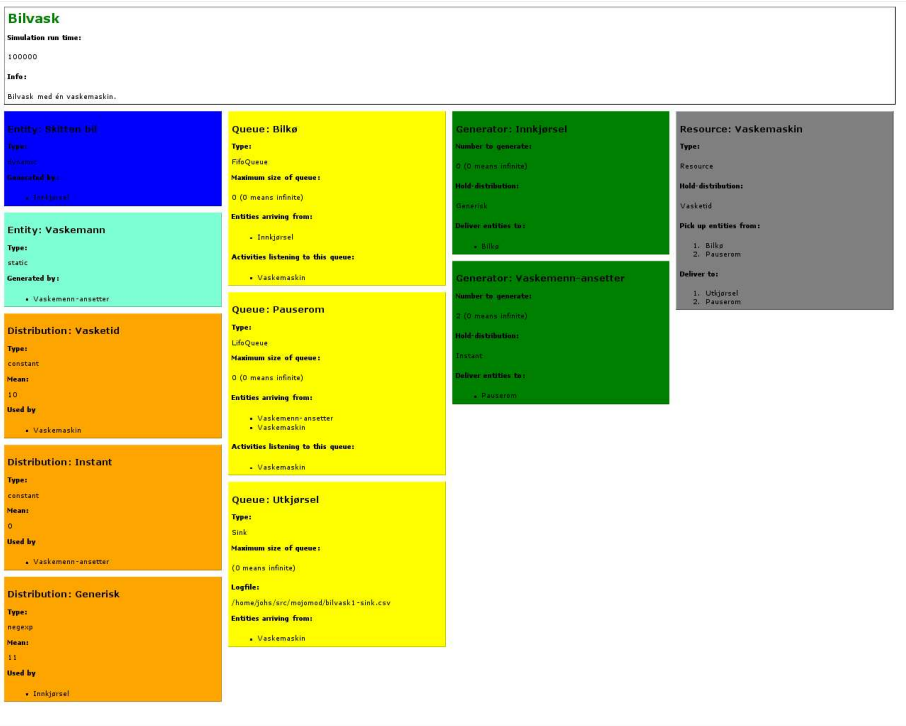
XSL, eller "utvidbart språk for stilark" (*eXtensible Stylesheet Language*) er et XML-basert språk for å definere stilark. XSLT er en delmengde av XSL og er et konverteringsspråk. Ved hjelp av dette definerer man en overgangsfunksjon fra et XML-dokumnet til et annet. Denne kan så kjøres av en XSL-motor sammen med et XML-dokument og generere et nytt XML-dokument.

XSLT er et *deklarerende* språk, imotsetning til Java, Simula eller C++, som er *imperative.[15]* Det innebærer at vi kan lage en mal på hvordan vi vil at resultatet skal bli og overlate resten til XSL-motoren.

Fra mjml til XHTML

Det vi har gjort er å lage en XSL-fil, mjml.xsl, som definerer en overgang fra mjml, som kan kjøres i mjmod, til XHTML, som kan vises i en nettleser.[5] Figur 8.5 viser en nettleser sin opptegning av en XHTML-fil generert av xsltproc gitt mjml.xsl og en mjml-fil som spesifiserer en bilvask.

[5] mjml.xsl er tilgjengelig på http://www.ii.uib.no/~mortene/mjmod/xhtml/mjml.xsl



Figur 8.5: Et eksempel på en modell i en nettleser.

8.3 Etterprosessering av statistiske data

En mer tradisjonell måte å sjekke i hvilken grad en modell er korrekt på er å gå gjennom de data simulasjonssystemet spytter ut. Her kan vi sjekke om vi mister enheter på veien, om noen resurser aldri får enheter osv. Dette kan kjapt gjøres også i vårt system ved å se på logfiler og utskrift til terminal under kjøring.

Del III

Utvalgte eksempler på modeller i mjml

Den beste måten å få innblikk i mulighetene til dette systemet er å se det i bruk, så i denne delen ser vi på noen utvalgte eksempler på modeller i mjml.

Vi har valgt et klassisk enkelt eksempel - bilvasken - for å vise modelleringstekniske detaljer ved å jobbe med mjmod. Utvidelsene vi kommer med til denne er mest for å vise frem funksjonalitet i mjmod ikke nødvendigvis aktuelle problemstillinger for en eier av en sådan vask. Vi vil av samme grunn ikke gå særlig dypt inn i analysen av resultatene av simuleringene.

De to siste eksemplene vi vil presentere er *Quality of Service* og en enkel heis.

"The proof of the pudding is in the eating."
- Miquel de Cervantes i *"The Adventures of Don Quixote"* [6]

[6](Rettere sagt i den engelske oversettelsen fra 1712 av Peter Motteux. Direkte oversatt fra spansk skulle det ha vært *"There will be laughter at the frying of the eggs"*.)

Kapittel 9

Bilvasken

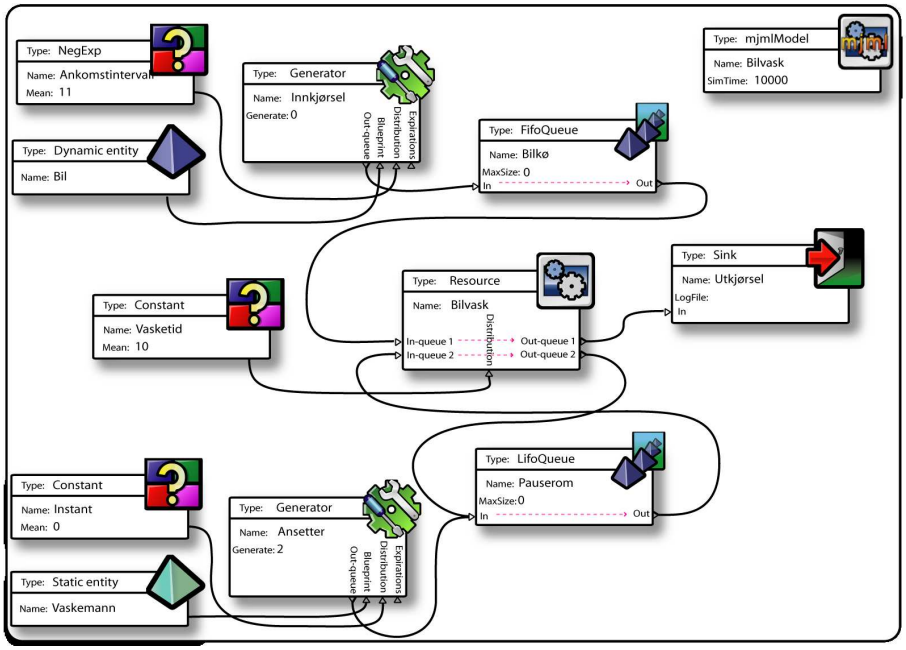
Bilvasken er et klassisk eksempel på en diskret hendelsessimulering. Vi tar utgangspunkt i en problemstilling av Helsgaun[13].

9.1 Bilvask med én vaskemaskin

Bilvasken med én vaskemaskin tar for seg et lite bilvaskeverksted med en innkjørsel, en kø av skitne biler som venter på å få bli rein og en vaskemaskin. Vaskemaskinen blir operert av vaskemenn som holder til på et pauserom i tider uten aktivitet. Slikt er enkelt å skape i ved hjelp av vårt simuleringsverktøy.

9.1.1 Logisk struktur

Først bestemmer vi oss for den logiske strukturen til modellen som er den beste tilnærmingen til problemstillingen vi vil simulere gitt de komponentene vi har til rådighet i systemet vårt. Strukturen kan se ut som figur 9.1.



Figur 9.1: Logisk struktur i bilvasken gitt komponentene i mjml.

Deretter starter vi opp et XML-redigeringsprogram og spesifiserer objektene som trengs. (Fullstendig mjml-kode finnes i appendiks D.4.1 på side 110.)[1]

Inspirert av Helsgaun sin bilvask bruker vi en negativt eksponensialfordelt tidsintervall mellom bilankomster med en median på 11, mens tiden vaskemaskinen bruker setter vi til konstant på 10 tidsenheter. Vi vil at *prosessene* kjøre 100000 tidsenheter, for så å la systemet tømmes for dynamiske enheter, så vi setter *maxtime* til 100000.

9.1.2 Grafisk tilbakemelding

XML-filen åpnes nå i mjmlGUI. Den tolker den og setter opp kjørbar modell samtidig som den eksporterer en grafisk representasjon av modellen i språket GraphXML. Figur 9.2 viser tilbakemelding til konsoll under tolking av en fil.

[1] Vi kan også gjøre dette i mjmlModeller, men vi velger å bygge mjml-filen i generelle XML-editorer, mjmlModeller foreløpig mangler *lagre*-funksjonalitet.

```
Parsing imports
Parsing distributions
Created a new distribution: Generisk(negexp:11)
Created a new distribution: Vasketid(constant:10)
Created a new distribution: Instant(constant:0)
Parsing entities
Created a new entity: Skitten bil(dynamic)
Created a new entity: Vaskemann(static)
Parsing queues
Created a new FifoQueue: Bilkø(maxsize:0)
Created a new LifoQueue: Pauserom(maxsize:0)
Created a new Sink: Utkjørsel.
Parsing processes
Created a new generator: Innkjørsel.
Created a new generator: Vaskemenn-ansetter.
Parsing activities
Created a new activity, Vaskemaskin a resource.
Finished parsed file
```

Figur 9.2: Tilbakemelding under parsing av en mjml-fil

Figur 8.4 viser hvordan grafen ser ut på skjerm når vi åpner den i Royere. Legg merke til hvordan ikonene hjelper til for å gjøre den logiske strukturen mer leselig.

Etter å ha tolket modellen kjøres den som spesifisert i 6. Systemet spytter kontinuerlig ut data til diverse filer for mer detaljert postprosessering, samt gir oss visuell tilbakemelding via det grafiske grensesnittet.

9.1.3 Statistisk tilbakemelding

De viktigste data kommer til terminalen like etter kjøring. (Figur 9.3)

```
Simulation: 'Bilvask'.
------------------------------
Simulation time elapsed: 100086.26588965867 units.
Maximum entities in the system: 31
Sink 'Utkjørsel' here.
I've gotten 9217 entities.
Average elapsed time: 64.06527068756947 units.
Maximum elapsed time: 309.9123020613479 units.
Minimum elapsed time: 9.999999999998181 units.
Resource 'Vaskemaskin' here.
I've served 9217 entities.
I've run 9217 times.
Maxtime through this resource: 10.0.
Busy for 92170.0 units of time
FifoQueue 'Bilkø' here.
Maxsize was 30.
0 entities expired before leaving this queue.
Average time in this queue: 54.06527068756938.
Maximum time in this queue: 299.9123020613479.
LifoQueue 'Pauserom' here.
Maxsize was 2.
0 entities expired before leaving this queue.
Average time in this queue: Infinity.
Maximum time in this queue: 63.719729758799076.
Size now: 2
'Vaskemann': Active 92170.0 units
'Vaskemann': Active 0.0 units
Generator: 'Innkjørsel' here.
I've generated 9217 of 'Skitten bil'.
0 entities were blocked due to full queues.
Generator: 'Vaskemenn-ansetter' here.
I've generated 2 of 'Vaskemann'.
0 entities were blocked due to full queues.
```

Figur 9.3: Utskrift av kjøringen av bilvask med en vaskemaskin

Første linjen av utskriften her i figur sier at simulasjonstiden ved simulasjonsslutt er 100086.26588965867, som jo er omlag 86 tidsenheter mer enn grensen vi satt til 100000. Det er fordi vi lar aktiviteter som enda har betingelsene sine oppfylt fullføre før vi blåser av simuleringen. I bilvask-eksempelet kan det sees på som at de bilene som allerede er kommet til køen blir vasket. Dette er jo gjerne ikke politikken til alle bilvasker, derfor kan man selvsagt kjapt endre på dette ved å sette *stop-unconditionaly-on-maxtime* til *sann* i modellspesifikasjonene.

Vi ser at det har på det meste vært 31 dynamiske enheter i simulasjonssystemet. I en såpass enkel modell som denne sier ikke dette så mye. Disse 31 vil nødvendigvis være 30 fra maksimal kø-lengde og én som er opptatt i vaskemaskinen.

Så kommer selve statistikkene fra bilene som har nådd utkjørselen. Vi ser at 9217 biler ble betjent, med en gjennomsnittstid på omlag 64 tidsenheter. Den

maksimale tiden en bil har brukt gjennom systemet er snaue 310 enheter, som jo må kunne regnes som en standhaftig sjåfør, da selve bilvasken bare tar 10. Minimumstiden er ikke så interressant i vår sammenheng, siden vi bare har en stasjon enhetene skal gjennom og den har en konstant tid. Da vil jo førstemann som blir plassert i køen (evt. enhver som blir plassert i en tom kø)[2] inn til denne stasjonen nødvendigvis den med kortest tid gjennom systemet.

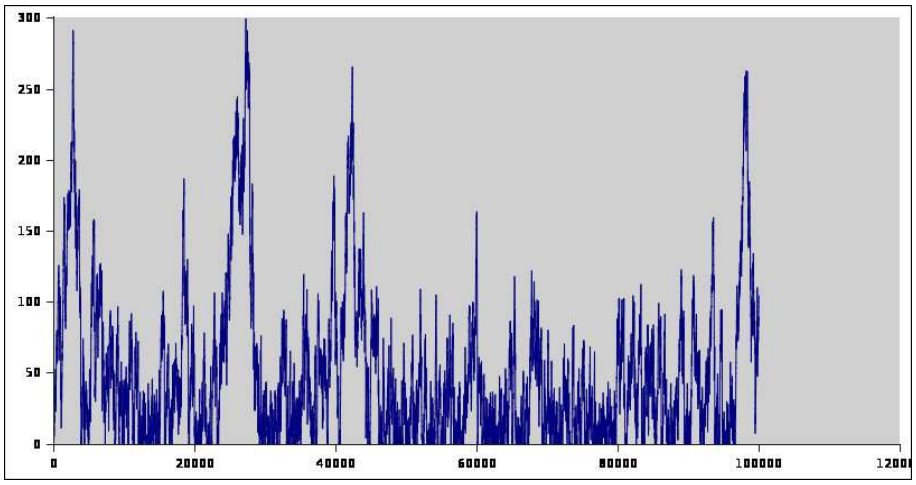
Deretter kommer litt info fra ressursene i systemet. Alle de 9217 bilene som nådde utkjørselen var innom denne vaskemaskinen vår og vaskemaskinen har gått like mange ganger. Dersom vaskemaskinen kunne tatt flere om gangen kunne disse verdiene vært ulike. Denne ressursen har vært aktiv i 92170 tidsenheter - altså et mål på ressursutnyttelsen.

Statistikken for køene følger så. Maksimal lengde på bilkøen er jo mindre interressant i dette enkle eksempelet - den kan jo leses rett fra maksimal tid brukt, men som vi skal komme tilbake til gir det mening i andre modeller. Til rapporten fra køen av vaskemenn har vi satt ordrikheten slik at vi får rapporter fra de enhetene som måtte befinne seg i den ved simulasjonsslutt. Vi ser at det bare er den ene vaskemannen som har vært aktiv - og da i like lang tid som vaskemaskinen. Vi legger også merke til at gjennomsnittstiden i denne køen rapporteres som uendelig. Dette er så siden det eksisterer enheter i den som aldri har forlatt den. Maksimumstiden i køen viser da hvor lenge en enhet maksimalt har oppholdt seg i køen før den har forlatt den. I dette tilfellet vil dette fortelle den lengste perioden vaskemannen har vært ledig og dermed bilvasken har stått uten å bli brukt.

Resten hopper vi over i denne omgang, men legger merke til at alle de 9217 bilene generatoren "Innkjørsel" har generert nådde utkjørselen .

Hver enhet stempler seg som sagt inn og ut i ulike køer og aktiviteter i sin vandring gjennom systemet. den informasjonen blir behandlet i utkjørselen og blir skrevet til en fil for post-prosessering. Figur 9.4 viser et plott av ventetiden i kø som funksjon av tiden i simulasjonen.

[2] I vårt tilfelle har vi i litt for mange gjeldende siffer og avrundingsfeil spiller inn og vi får tilsynelatende en enhet som har brukt mindre tid gjennom systemet enn strengt tatt skulle være mulig. Dette kan enkelt løses ved å innføre færre gjeldende siffer, men grunnen til at vi ikke har valgt å sette antall gjeldende siffer i listen med resultater er at vi vil at systemet skal være så generelt som mulig og at det bør være opp til brukeren å sette antall siffer ut fra parametrene i systemet. Vi kan evt. i en revidert utgave av mjml spesifisere det i mjml-filen.



Figur 9.4: Ventetid som funksjon av tiden i en bilvask med én vaskemaskin.

Som vi ser svinger ventetiden temmelig mye, men stort sett klarer maskinen å svelge unna.[3] Det blir ingen vedvarende opphopning av biler i køen. Vi ser derimot at mange av ventetidene er temmelig lange i forhold til vasketiden, så hva hender om vi innfører en grense på bilkø-lengden?

9.2 Bilvask med én vaskemaskin og maksimal kølengde på 10

La oss si at det bare er plass til 10 biler i køen hos bilvaskeren, som er mer realistisk enn uendelig. Dette er lett å forandre på, vi endrer følgende linje i spesifikasjonen av bilkøen:

```
<maxsize>0</maxsize>
```

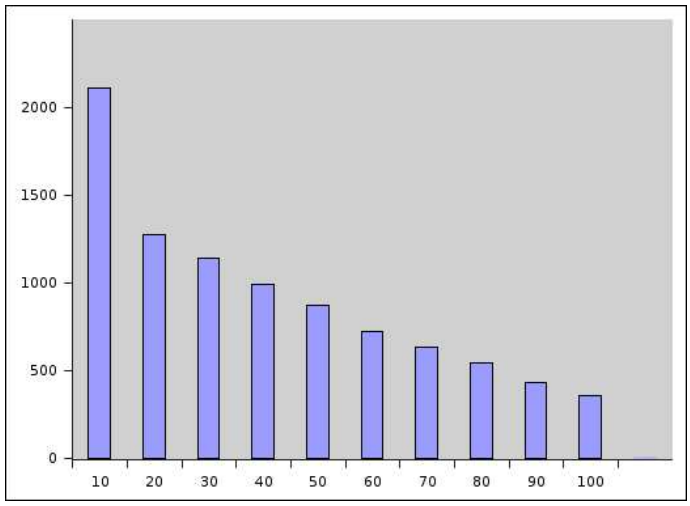
til:

```
<maxsize>10</maxsize>
```

og kjører simulasjonen igjen med resten av oppsettet likt.

Da ser vi at 139 av disse 9217 bilene må avvises og gjennomsnittlig tid brukt gjennom systemet synker fra 64,05 til 45,33 tidsenheter. Så kan se på histogrammet (Figur 9.5) over bilene, som gjerne sier mer interressant enn forrige graf, da den sier mer om hvor lenge biler egentlig venter i systemet.

[3]Bemerking: Eneste fordelingen som er inne i bildet her er en negativt eksponentialfordeling, som er uten hukommelse, så det at vi kan se konturene av en slags syklisk oppførsel på systemet er nok tilfeldig.

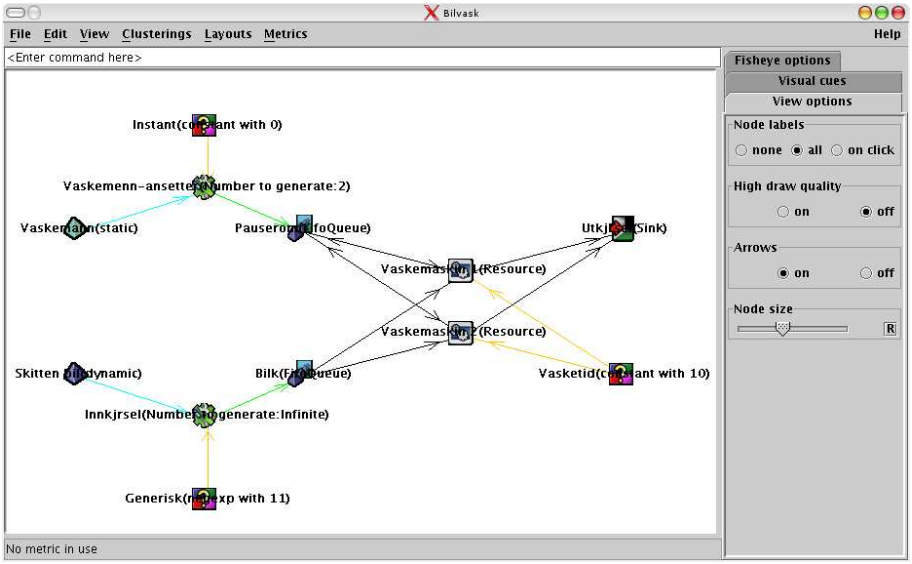


Figur 9.5: Histogram delt opp i ventetider i en bilvask med én vaskemaskin med maksimal kølengde på 10.

Fortsatt er ventetiden temmelig lang, så vi kan prøve å korte den ned ved å sette opp en ekstra vaskemaskin og se hva som hender da.

9.3 Bilvask med to vaskemaskiner

Å sette opp en ekstra vaskemaskin er enkelt gjort i XML-filen vår; en enkelt klipp-og-lim-operasjon og vi har to vaskemaskiner. Eventuelt kan vi legge til en ekstra ressurs i mjmlModeller som har pekere til de samme køene som den allerede eksisterende vaskemaskinen. Vi lar begge vaskemaskinene hente biler og vaskemenn fra de samme køene. Figur 9.6 viser hvordan strukturen ser ut i grafen vår.



Figur 9.6: Bilvask med to vaskemaskiner i Royere

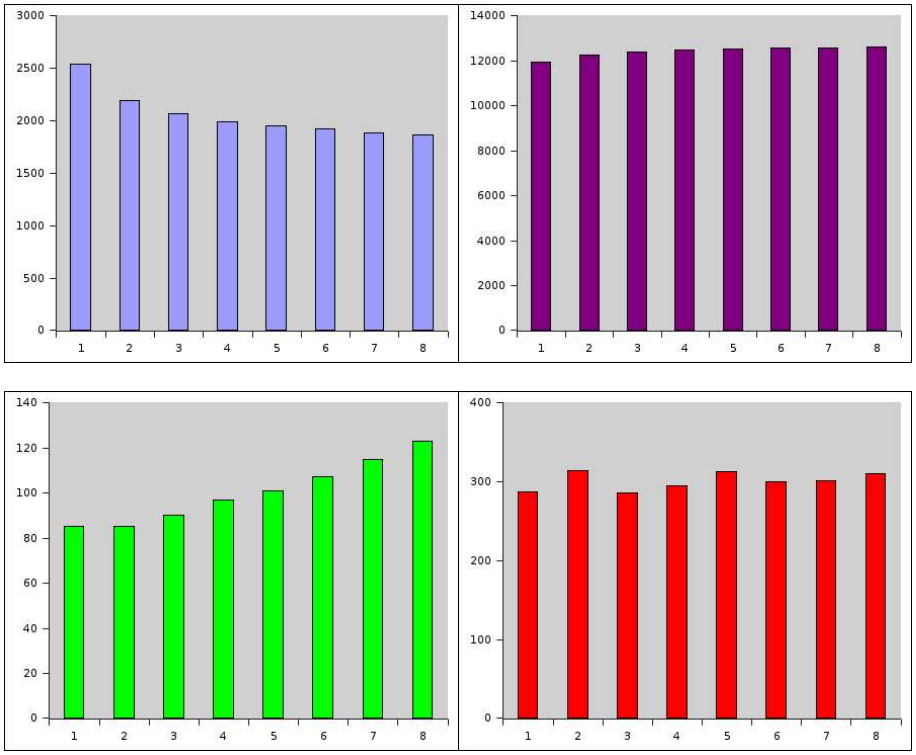
Med to vaskemaskiner klarer bilvasken å betjene alle de 9217 kundene. Vår prioriterte vaskemaskin nummer 1 tar seg av 5603 kunder, mens den nyervervede vaskemaskin 2 tar seg av 3614 av dem. Vi ser at gjennomsnittstiden gjennom systemet er nede i 11,42 tidsenheter, det vil si at gjennomsnittlig ventetid i kø er knappe 1,42, mens maksimumstiden fra innkjørsel til utkjørsel er på 38,52 tidsenheter.

Vaskemennene har delt temmelig broderlig på arbeidet med vaskingen; den ene aktiv 46400 og den andre 45770 tidsenheter.

9.4 Bilvask med to vaskemaskiner og en betalingsstasjon

Det neste scenarioet vi kan tenke oss er at vi setter opp en betalingsstasjon etter de to bilvaskene. Det er plass til en begrenset mengde biler i køen til denne, og dersom køen er full får ikke vaskemaskinene gjort jobben sin og biler blir stående i vasken.

Dette implementerer vi i modellen vår ved å legge til en kø kalt betalingskø samt en ressurs kalt betalingsstasjon. Vi tenker oss at gjennomsnittlig tid per betalingsprosedyre er 8 tidsenheter. Det vi gjerne lurer på er hvor mye lengden på køen spiller inn på hvor mange biler vi klarer å betjene på travle dager på vaskestasjonen. Vi justerer da ned medianen til fordelingen av biler inn i systemet til, la oss si, 7.



Figur 9.7: Resultater fra bilvask med to vaskemaskiner og en betalingsstasjon

I figur 9.7 ser vi resultatene av vårt lille eksperiment. X-aksen i grafene gir kølengde til betalingsstasjonen, Y-aksen sier (med klokken fra øverst til venstre) antall blokkerte biler, antall serverte biler, gjennomsnittstid gjennom systemet og maksimumstid gjennom systemet.

9.5 Bilvask med en vaskemaskin og biler som gir opp

Et annet scanario man kan tenke seg er at biler "gir opp" køen etter at de har stilt seg i den. Dette oppnår vi ved å definere et fordelings-objekt og koble det til generatoren for biler om en fordeling for utløpstider (*expiration distribution*), slik:

```
<expiration-distribution>Gi opp tid</expiration-distribution>
<expiration-distribution-pointer>4</expiration-distribution-pointer>
```

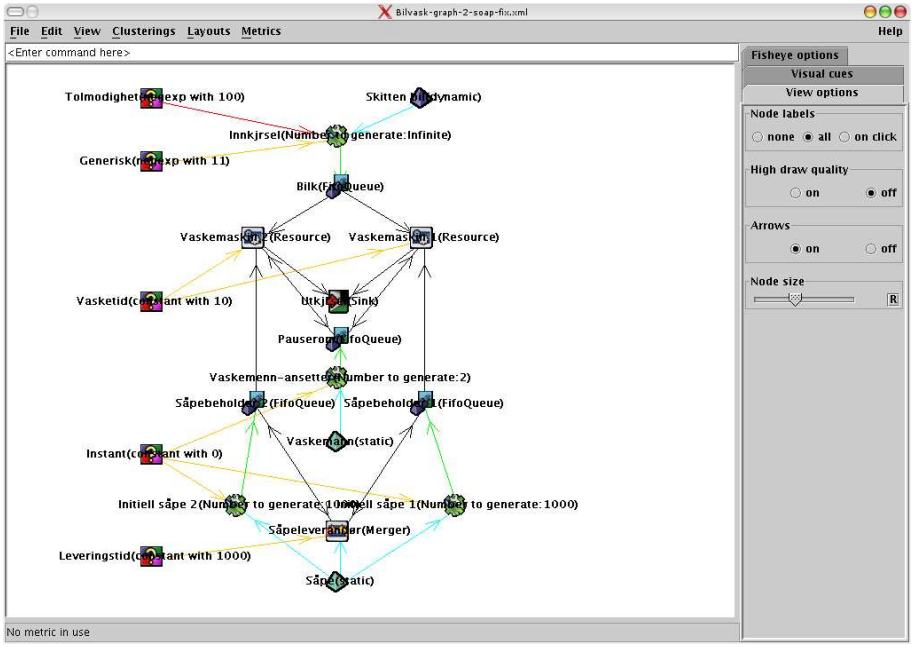
Denne distrubusjonen spesifiseres som en hvilken som helst annen fordeling. Konsekvensen av å koble på denne fordelingen er at vi nå setter et flagg på biler som sier hvor lang tid det vil ta før de gir opp og forlater systemet. I dette tilfellet tenker vi oss at tidene for å gi opp er negativt eksponesialfordelt med

en median på 13. Vi beholder grensen på 10 biler i køen[4], kjører simuleringen og får følgende resultater:

Gjennomsnittstiden på de enhetene som kommer gjennom systemet synker til 10,59 enheter, med en maks-tid på 19,81. 476 av av disse 9217 genererte enhetene gav opp å vente på ledig vaskemaskin. Altså fikk vi 8741 kunder gjennom. Maksimal lengde på køen var 5 og det var altså ingen som kjørte forbi uten å stille seg i kø.

9.6 Bilvask med to vaskemaskiner og begrensede såpebeholdere

Andre begrensninger vi kan forestille oss i en bilvask er tilgang på såpe. Vi spesifiserer først separate såpelagre for hver bilvask. Vi vil gjerne ha en viss mengde såpe i systemet ved start så derfor setter vi opp to generatorer som ved simulasjonsstart fyller beholderene med en mengde såpe. Videre benytter vi en ny form for aktivitet; en merger. En av dens egenskaper er at den kan lytte på vilkårlig mange køer og kjøres når en av disse er tom. Det den da gjør er å fylle en gitt kø med gitt antall av en enhet den genererer ut fra et blåpapir, ala en generator, men i motsetning til generatorer kan den generere alle på en gang. Dette er nødvendig for denne modellen her. Resten av modellen er lik "Bilvask med to vaskemaskiner" 9.3 på side 67.



Figur 9.8: Skjermbilde av bilvask med såpebegrensing visualisert i Royere

Vi setter initiell såpemengde i begge såpebeholderene til 100, så vaskemask-

[4]selv om til en viss grad disse to har samme "effekt" på denne enkle modellen

inene har noe å starte med. Hver gang såpeleverandøren trengs bruker den 100 tidsenheter og den leverer 60 såpeenheter av gangen. Etter å ha kjørt modellen ser vi at bilvasken nå må avvise 5 biler siden køen inn til vaskene har vært full, gjennomsnittstid gjennom systemet øker fra 11,42 til 14,26, og maksimumstiden går fra 38,52 til 123,82 enheter. Dette vaskesystemet tilsvarer jo et system der såpelageret er nærme bilvasken, men vi kan jo tenke oss at når først går tomt for såpe tar det temmelig lang tid å hente ny. La oss si at vi starter med et såpelager på 1000, for så å få nye 1000 hver gang vi går tom, og denne leveransen tar 1000 tidsenheter. Klarer da to bilvasker med separate såpebeholdere å ta unna bilene fra isted, eller vil de ende opp hos konkurrenten ved siden av?

Svaret er nesten; 74 biler kjørte forbi da køen var for lang. Gjennomsnittstiden var omlag den samme, men makstiden ble økt til 833,06, noe som kan tyde på at begge bilvaskene var tomme for såpe på samme tid. 833 tidsenheter er igjen litt lenge å vente så vi innfører en "utløpstid" på disse bilene, for å få et mer realistisk overslag over hvor mange kunder vi vil miste. Vi setter opp en slik som en negativt eksponensialfordelt fordeling med en median på 100. Da ser vi at 201 biler gir opp å vente på vasking, gjennomsnittstiden gjennom systemet er nede i 11,78 og maksimumstiden er 167,12.

9.7 Resultater sammenlignet med Helsgaun sin javaSimulation

Et av eksemplene til Helsgaun i [13] er nettopp bilvasken med variabelt antall vaskemaskiner. Vi kjørte hans pakker for å se hvilke resultater det gav, og om det var på noen måte sammenlignbart med våre.

9.7.1 Ikke elementært å sammenligne to simuleringspakker

Nå er det ikke elementært å sammenligne to diskrete hendelsessimuleringssystemer slik, da ett sett av tilfeldige tall kan gi et helt annet utslag enn et annet sett og dermed gjøre slikt vanskelig selv i en liten modell som dette. Det vi derimot benytter oss av er det faktum at vi har bygget vår tilfeldige-data-generator på en lest spesifisert av nettopp Helsgaun sin simulator, og det at gitt samme initialverdi (*seed*) gir den alltid samme tallrekke ut. Dermed vil rekkefølgen på hendelsene i hans system samsvare med vårt.[5]

9.7.2 Resultater

Han velger å la systemet sitt kjøre i 1000000 tidsenheter, så vi kjører vårt system om igjen med et slikt tidsintervall. Først med én bilvask. Da får vi 90938 biler gjennom systemet, en maksimal kø-lengde på 42 og gjennomsnittstid på 60,55 enheter. Med 2 bilvasker får vi fortsatt 90938 biler gjennom systemet, maksimal kø-lengde er på 7 og gjennomsnittstiden er på 11,37 enheter. Så kjører vi Helsgauns tilsvarende systemer og får resultatene i figur 9.9.

[5] Dette gjelder i enkle tilfeller som dette, men i mindre grad i tilfeller der prioriteringer på aktiviteter osv spiller inn.

```
1 car washer simulation No.of cars through the system = 90942
Av.elapsed time = 61.35
Maximum queue length = 45
2 car washer simulation No.of cars through the system = 90942
Av.elapsed time = 11.42
Maximum queue length = 7
```

Figur 9.9: Utskrift av Helsgaun sin bilvask-simulator

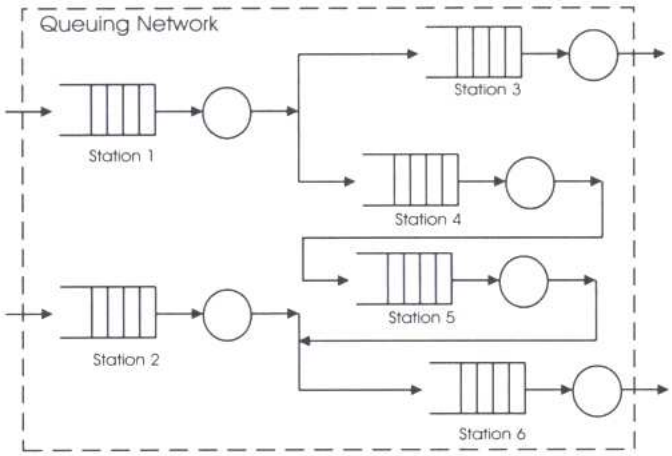
Det første vi legger merke til er at Helsgaun sitt system sender flere biler gjennom. Ikke mange, men 4. Det er fordi den implementasjonen vi kjørte her er hendelses-orientert og han løser "problemet" med avsluting av simulasjonen på en litt annen måte enn oss. Denne observasjonen kan også gi oss grunnen til at maksimal kølengde i hannes system er 3 mer enn i vårt. Opphopningen ved vaskemaskinen er gjerne på sitt største ved slutten av simulasjonen og dermed vil eventuelle sist-ankomne være til å skape den lengste køen i systemets historie. Vi ser iallefall at størrelsesordenene er de samme.

Dette er fortsatt temmelig enkle modeller av diskret hendelsessimulering., men hva om bilvaskeren opererer med mange ulike programmer for bilvask, og gjerne flere innkøer? Da kan man tenke seg et nettverk av køer.

Kapittel 10

Kø-nettverk

Garrido viser et generisk eksempel på et nettverk av køer. Et slikt nettverk består av et vilkårlig antall sammenkoblede stasjoner. Stasjonene er koblet slik at det den ene stasjonen spytter ut, blir innputt til en annen. Figur 10.1 er hentet fra Garrido[12].



Illustrasjonen er hentet fra Garrido[12].

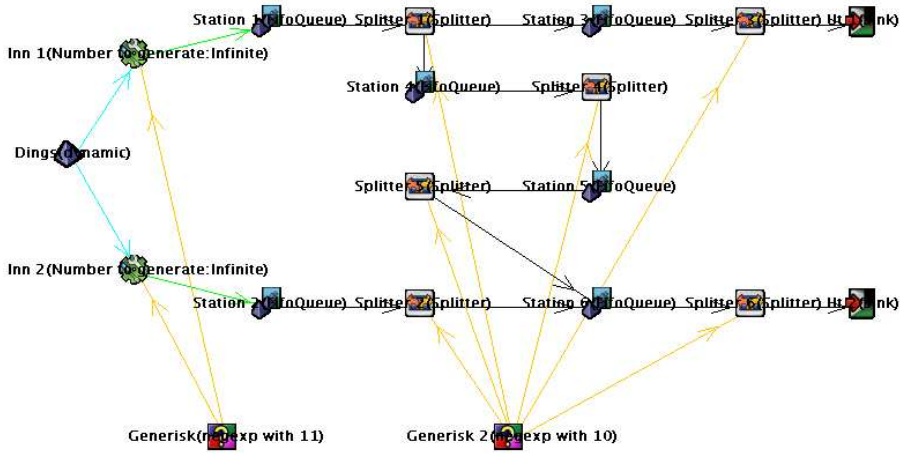
Figur 10.1: Logisk struktur i kø-nettverket.

For å holde oss til verden av bilvasker litt til, kan dette sees på som en bilvask med to innganger og ulike rekkefølger på ulike programmer. Biler som kommer inn inngang 1 vil ha program 1 for så å velge mellom program 3 eller en serie av programmer 4, 5 og 6. Inn inngang 2 kommer biler som vil ha program 2 og 6. Garrido bruker dette eksempelet som et mer abstrakt system som kan sees på feks. som trafikk gjennom butikk-kasser eller lignende.

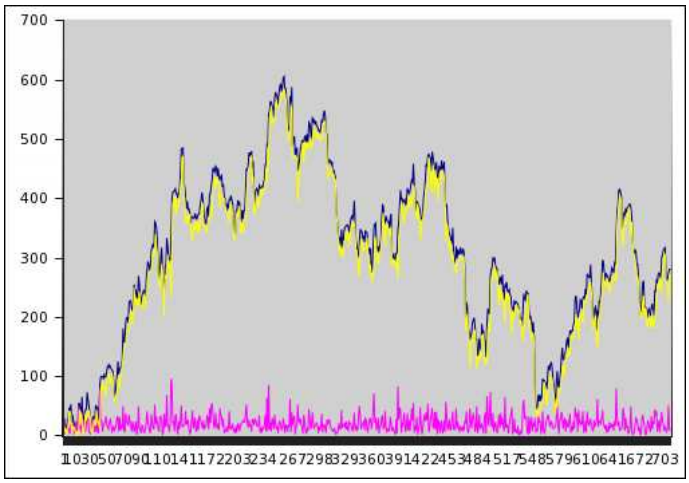
Et annet eksempel kan være et bryggeri sitt flasketappesystem. Inn på kø 1 kan vi ha skitne flasker. De kan bli sortert etter om de er hele eller skadede. De som kasseres kan plasseres på stasjon 3 mens de som får en ny sjanse blir

plassert på stasjon 4 for kø til vasking. Stasjon 5 kan for eksempel være kø for tørking. På stasjon 2 kan allerede reine flasker ankomme. Fra både stasjon 2 og 5 går veien til tapping på stasjon 6.

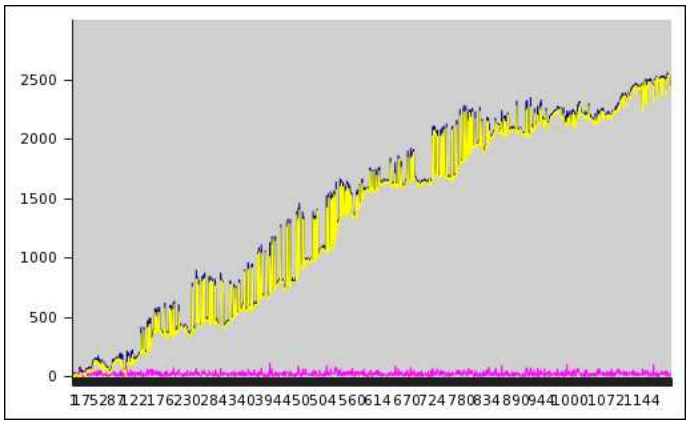
Vi implemeterte dette i mjml(D.4.2 på side 113). Grafen som ble eksportert av systemet vårt ser vi visualisert i Royere på figur 10.2. Legg merke til at vi bare trenger å legge til generatorer, utslagsvasker, fordelinger og en blåpapirsenhet for å spesifisere hele modellen grafisk.



Figur 10.2: GraphXML-visualisering av kø-nettverket vårt.

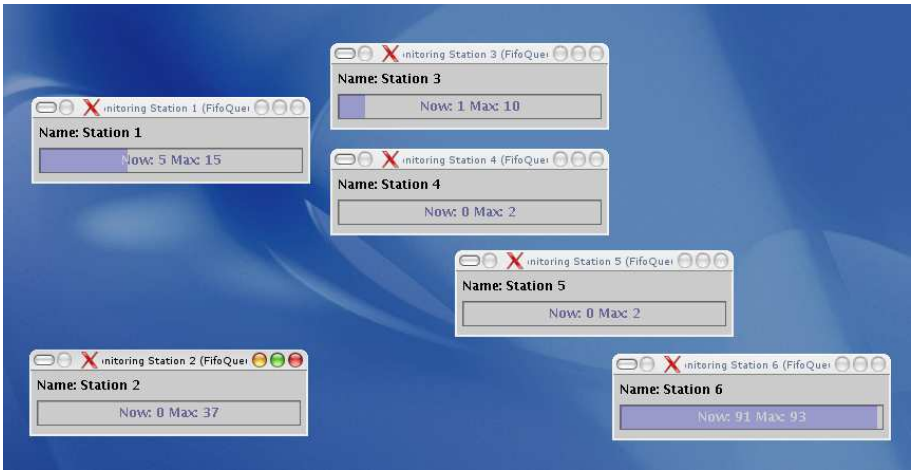


Figur 10.3: Graf over tider gjennom kø-nettverket til utgang 1



Figur 10.4: Graf over tider gjennom kø-nettverket til utgang 2

Siden ikke Garrido hverken gir eksempler på kjøring av systemet eller spesifiserer fordelinger i sin utdypning av kønettverket har vi ikke mulighet til å sammenligne data med hans kjøring. Grafene i figur 10.3 og 10.4 blir her da mer som eksempler på statistikksamling i systemet. Vi legger merke til at enheter bruker stadig lengre tid på å nå utgang 2. Det foregår altså en opphoping i systemet.



Figur 10.5: Sanntidsovervåking av stasjonene i kø-nettverket

Vi starter det grafiske sanntidsgrensesnittet og velger å overvåke stasjonene 1 til 6 for å se hvor denne opphopningen skjer. Figur 10.5 viser dette i praksis. Her har vi lagt ut de ulike monitorene løselig basert på hvordan de henger sammen i modellen. Vi ser dermed at opphopingen skjer på stasjon 6.

Kapittel 11

QoS i et datanettverk

Hvilke pakker når frem og hvilke blir stoppet i et QoS-rutesystem i Linux-kernelen?

11.1 QoS i et nøtteskall

QoS (Quality of Service), i datanettsammenheng, er kort forklart et verktøy for å prioritere ulik netttrafikk ulikt. Altså i en internettforbindelse vil man typisk minimere latensen ("tid fra man ber om noe til noe hender") på sanntidstjenester som telnet, ssh, vnc og spill. Nettsurfing og epost kan prioriteres like under disse mens FTP- og p2p-/fildelingstrafikk kan godt prioriteres lavest (noe som tar en dag å laste ned kan like godt ta litt mer tid).

11.2 Scenario

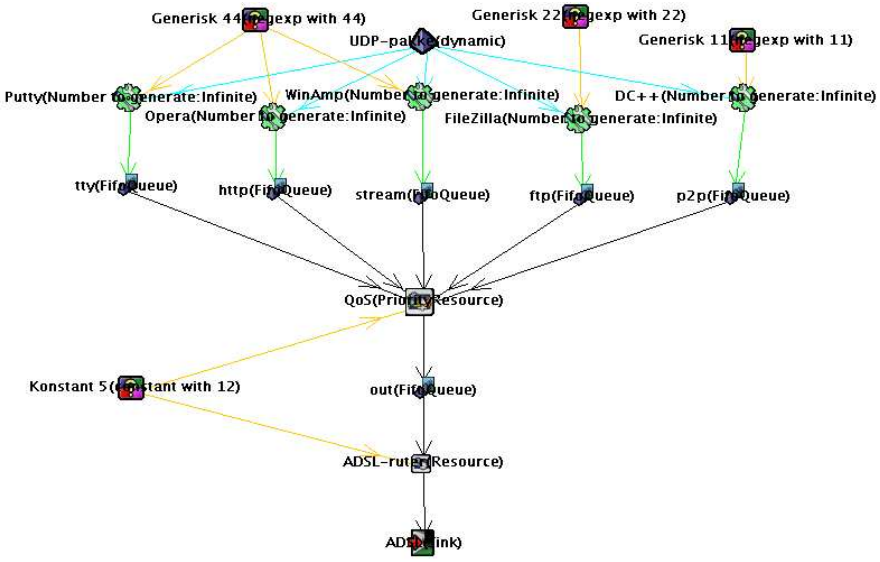
Vi kan tenke oss følgende scenario: Seks aktive nettbrukere deler en asynkron linje til nettet; ADSL, med omlag 2Mbit per sekund inn og 600kbit per sekund ut. Forenklet kan trafikken disse genererer deles opp i pakker som kan variere i størrelse opp til 1500 bytes. Begge sider av forbindelsen, leverandør (ISP) og sluttbrukerene, har til enhver tid hvert sitt buffer (organisert som en FIFU-kø) av pakker som er klar for å sendes gjennom linjen. Dersom linjen er helt saturert av trafikk vil vi naturlig nok få stor latens på trafikken da alle pakker må vente i denne køen på tur og den kan være flere sekunder lang. Dette merkes best på sanntidstjenester (interaktive tjenester), der man sitter hele tiden og venter på at systemet skal gi tilbakemelding på enhver innputt man selv gir.

Vi vil fokusere på den delen av nettet vi som sluttbruker har tilgang til å justere ved hjelp av QoS, nemlig linjen ut. Køen av pakker på vei ut ligger i ADSL-ruteren. Dersom den er full mister man pakker, de må sendes på nytt og saker og ting får enda mer latens. Det vi altså vil gjøre er å flytte køen til et sted vi når den, nemlig en linux-ruter, slik at vi har kontroll over den. Dermed kan vi levere pakker til ADSL-ruteren i en hastighet som ligger litt under ut-hastigheten. Køen ut kan vi dermed kontrollere selv og la pakker vi mener er viktigere enn andre pakker gå først i den.

11.3 Modell

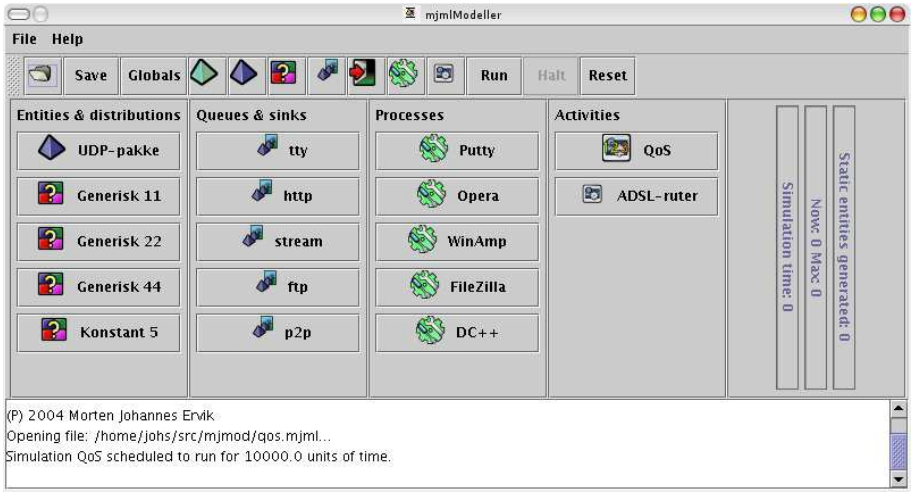
Siden QoS, og ikke minst datanettverk, er en vitenskap i seg selv og utenfor omfanget på denne oppgaven skal vi ikke gå i dybden på resultater i forbindelse med denne modellen, men bare skissere en løsning.

Figur 11.1 viser hvordan vi tenker oss den logiske strukturen på modellen. Sentralt er prioritetsressursen som alltid henter enheter fra den køen med høyest prioritet som har noen å gi. Generatorene representerer data-programmer som genererer ulik form for nettverkstrafikk.



Figur 11.1: QoS i Royere

Figur 11.2 viser modellen lastet i mjmlModeller klar for kjøring og/eller redigering.



Figur 11.2: QoS i Royere

Kapittel 12

Heisen

(Illustrasjon fra: http://www.elevatormuseum.com)

Figur 12.1: En tidlig heis fra 1866.

En annen klassisk modell i diskret hendelsessimulering er heisen. Denne står beskrevet blant annet i Donald Knuth sin *"The art of Computer Programming"* fra 1969[17]. Dette er et ikke helt elementært stykke diskret hendelsessimulering, da svært mange avhengigheter inngår. Figuren viser hvordan hendelsene i heisen henger sammen - altså tilnærmingen hendelses-planlegging.

12.1 Problemstillingen

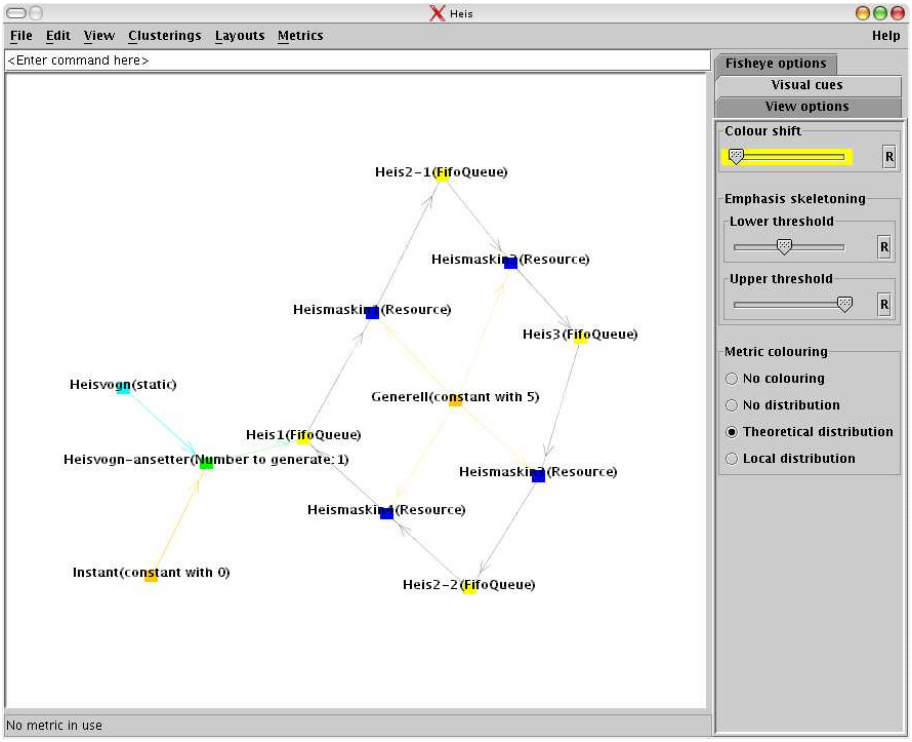
Vi kan vi tenke oss følgende utvalg av diskrete hendelser i en slik heissimulering for en person gjennom systemet:

1. Personen Per genereres.
2. Per ankommer heisen i 3. etg og trykker på heisknapp.
3. Knappetrykket når heisen som er i 1. etg og den "bestemmer seg" for å hente nyankomne siden den ikke har noe bedre å finne på.
4. Heisen lukker dørene sine.
5. Heisen akselererer i retning av etasjen der knappen ble trykket.
6. Heisen beveger seg opp en etasje - ingen har trykket på knappen der.
7. Heisen beveger seg opp enda en etasje - der har Per trykket på knappen.
8. Heisen stopper og åpner dørene sine.
9. Per går inn i heisen.
10. Per angir måletasjen 1. etg.
11. Heisen lukker dørene sine.
12. Heisen akselererer i retning av valgte mål - ned.
13. Heisen går ned en etasje. Ingen ned-knapp trykket, der.
14. Heisen fortsetter ned en etasje - og når valgte måletasje.
15. Heisen stopper og åpner dørene sine.
16. Per kan forlate heisen (og systemet).

12.2 Heisen modellert i mjmod

For å forenkle heisscenarioet litt tenker vi oss at heisvognen går kontinuerlig opp og ned, selv om ingen personer venter ved noen av etasjene. Dessuten tenker vi oss at vi bare har 3 etsasjer i huset. (Siden skal det bare en klipp og lim-jobb til for å få så mange etasjer i huset som datamaskinminnet vårt tillater.) Det første vi setter opp er selve representasjonen av heissjakten. På figur 12.2 ser vi hvordan vi tenker oss en rettet graf der vi har en heismaskin (ressurs) mellom hver etasje (kø) som flytter en tenkt heisvogn fra etasje til etasje.[1]

[1] En annen ting som er verdt å merke seg med grafen i figur 12.2 er at her har Royere tatt seg av all utlegging av grafen. Dette er en algoritme kalt *"GEM-Force-Directed"*, og fungerer bra når grafene er såpass små som dette. Dessuten har vi her valgt å ikke bruke ikoner i GraphXML-filen, men bare fargekoding. Dette kan benyttes om grafer blir for uoversiktlig med ikoner.



Figur 12.2: Heissjakten i Royere uten ikoner, men med fargekoding.

Vi tenker oss at vi genererer en heisvogn-enhet ved tid 0 og plasserer den i heissjakt-systemet. Der vil denne svirre og gå til vi stopper den, om vi kjører denne modellen. Dette er ikke den heisvognen som vi kommer til å plassere personer som går gjennom systemet i, men snarere en slags teller som holder styr på hvilken etasje heisen til enhver tid er i og hvilken retning den har.

12.2.1 Tagger fungerer som "ønsker"

Resten av systemet gjør bruk av en tagg-finesse i systemet vårt. Vi fester tagger på personene som går inn i systemet, slik at aktiviteter kan utføre forskjellige ting ut fra denne merkingen siden. Vi tenker oss altså at personer blir generert et sted og så får utlevert tagger av "taggskrivere" *(TagWriter)* som sier hvilken etasje de skal reise fra og hvilken de skal reise til. Disse taggskriverene bruker et helt vanlig fordelingsobjekt til merkingen, noe som gjør at de kan hente tagger fra fil om man vil kjøre en modell med data fra et virkelig heissystem. Vi lar dem skrive tilfeldige tall mellom 1 og antallet etasjer i huset på taggene. Etter to slike merkinger, en for inn-etasje og en for ut-etasje, kommer vi til en Splitter (som tar enheter fra én kø og plasserer dem ut i vilkårlig mange andre). Dette er samme redskapet som i kø-nettverket, men denne gangen splitter den etter tagger. Enheter den skal splitte må ha blitt gitt taggen den leter etter, i dette tilfellet "inn". Så plasseres enhetene i sine respektive etasje-køer og venter der på heisen.

12.2.2 Inn i heisen

(Minst) én aktivitet sitter i hver etasje og lytter etter personer i inn-køen i denne etasjen samt etter om heisvognen er i heissjakten her. Legg merke til at hver etasje utenom topp- og bunn-etasjen må ha to aktiviteter som lytter, da vi må ta hensyn til om heisen er på vei opp eller ned. Dersom heisen er i heissjakten på aktivitetens etasje og det er folk som venter på heisen holder dem opptatt tiden det tar å gå inn i heisen ved å ta dem til seg (den kondisjonelle fasen) og holde dem en tid diktert av fordelingsobjektet dens. Etter denne tiden har gått plasserer aktiviteten heisen tilbake i heissjakt-representasjonen og personen i heisen.

En ting som vi kan tenke oss kunne ha hendt, i et mer generelt eksempel, er at heisvognen kunne blitt full i mellomtiden, og dermed ikke aktiviteten kunne levert fra seg objektene sine.[2] Dette problemet omgås ved å sette "sprett ved ingen leveranse"-biten *(bounce-on-no-delivery)* til aktiviteten, som gjør at dersom den ikke klarer å levere sakene sine, stiller den dem tilbake der den hentet dem fra og venter litt (*wait-on-no-delivery*, eller *wait*) før den kan få oppfylt sine betingelser igjen, slik at andre aktiviteter kan bli prioritert.

12.2.3 Aktivitetenes prioriteter

Prioritet på aktivitetene er et annet nøkkelord i heissimuleringen vår. Her er det feks. kritisk at disse etasje-aktivitetene er prioritert over de aktivitetene som flytter heisenvognen i sjakten, ellers vil bare heisen gå evig uten at andre får sleppe til. Dette løses ved at aktivitetene som er definert først har høyest prioritet. Det kan gjerne løses på andre, mer elegante måter, men det er utenfor omfanget til denne oppgaven. Vi definerer også resursene som henter personer ut av heisen før aktivitetene som plasserer folk inn i heisen for å simulere at folk slepper ut av heisen før nye kommer inn.

12.2.4 Ut av heisen

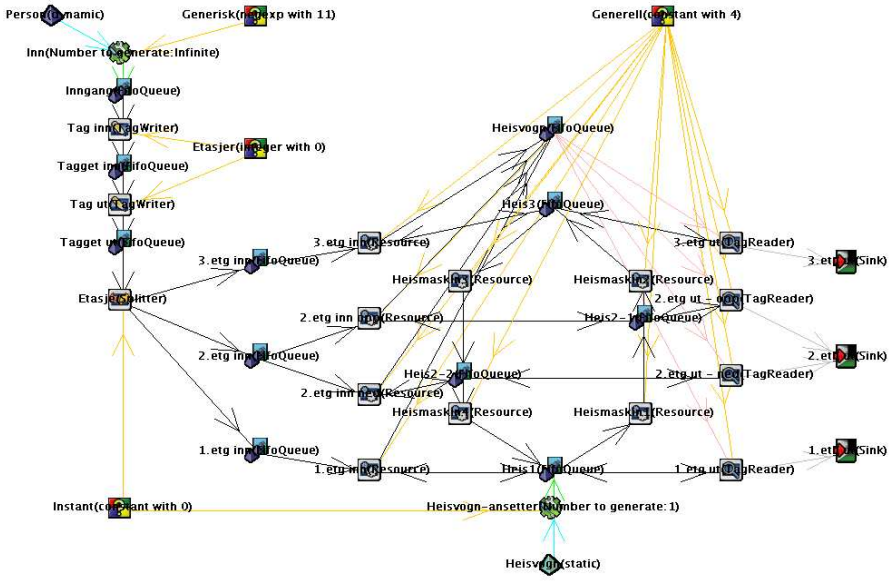
Motpartene til disse etasjeaktivitetene er de aktivitetene som henter ut personer som har kommet til rett etasje. De er tagglesere (*TagReader*) og leter etter "ut"-tagger som stemmer over ens med det etasjenummeret de står i. De ser etter om heisen er i sjakten ved deres etasje. Om ja; skanner de heisvogen etter første person med "ut"-tagg som passer, dersom det er ingen som har det, gjør den ingenting. Om de finner noen holder dem denne personen og heisobjektet den tiden det tar å forlate heisen, tilsvarende til fra-etasjen. Dette modellerer tiden det tar for en person å forlate heisen.

Siden disse taggleserene i etasjene har høyere prioritet enn heisflytterene vil denne prosessen gjentas helt til det ikke lenger er personer i heisen med taggen taggleseren leiter etter, og heisen vil atter bevege seg. Her har vi ikke satt en "sprett ved ingen leveranse", da vi kan tenke oss at dersom en person ikke kommer seg ut av heisen pga at det er fullt utenfor, ja så vil nok heisen stå i ro til denne klarer å komme seg ut. taggleserene plasserer personene i utslagsvasker, svarende til ut-etasjer, for statistikk-henting.

[2] I vårt eksempel er det bare en etasje av gangen som *kan* fylle opp heisen, så her trenger vi strengt tatt ikke denne funksjonaliteten. Vi kunne benyttet oss av å sjekke ut-køen(e) som en del av betingelsene for å starte aktiviteten, men det gjør vi ikke da vi vil at heisen skal stoppe opp i en etasje der noen ønsker å gå på, uavhengig om det er plass til dem eller ikke.

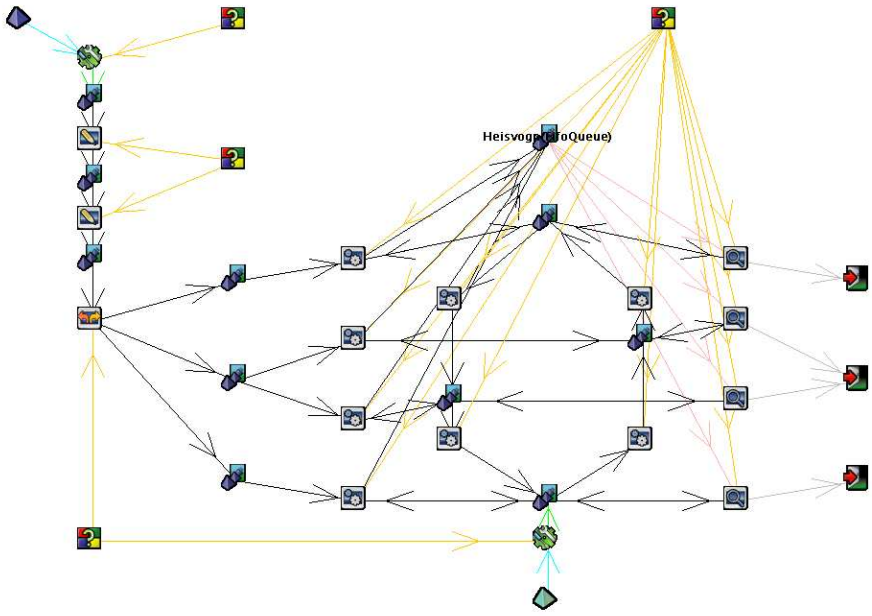
12.2.5 Grafisk oversikt over heisen

Figur 12.3 viser hvordan heisgrafen vi har laget ser ut i Royere med ikoner.



Figur 12.3: Hele heisen i Royere

Her kommer Royere sin funksjon for å slå av node-navn til sin rett, da blir det som i figur 12.4. Da ser man strukturene uten at det blir for rotete. Ved å markere en node får man frem navnet dens igjen. Fargekodingen vår kommer da også til sin rett. Andre finesser Royere har å by på for uoversiktelige grafer er et "fiskeøye-syn" med varierende grader av fiskeøye-effekt og opp til 3 ulike fokuspunkter. Dette får grafen til å bule ut noen steder og trekke seg sammen noen steder og man kan på den måten få oversikt over deler av grafen som gjerne kan være vanskelig å få oversikt over ellers.



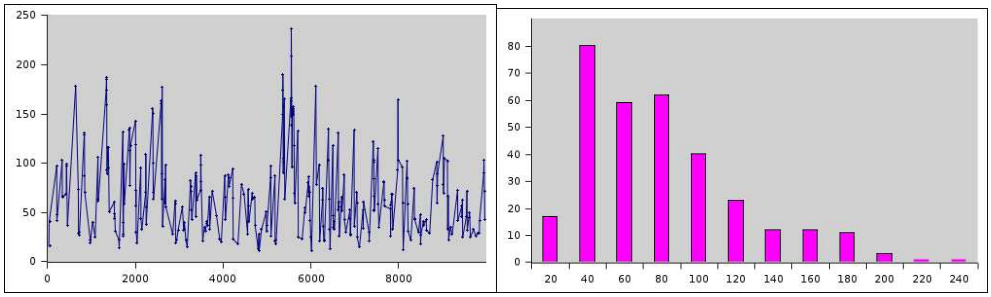
Figur 12.4: Hele heisen i Royere, uten nodenavn

12.3 Resultater fra heissimuleringen

Så vil vi gjerne finne ut hvor lang tid det gjennomsnittlig tar å komme seg til ønsket etasje i huset, eller hvor stor heisen må være for å svelge unna en gitt mengde personer, eller lignende.

Vi tenker oss først at vi vil se hvor mange heisen må ta dersom det materialiserer seg en person med ønske om å kjøre heis i gjennomsnitt hvert 11. minutt negativt eksponentialfordelt. Tiden det tar for heisen å bevege seg mellom etasjer, samt tiden det tar å gå inn eller ut av heisen setter vi til 4 tidsenheter. Etter å ha kjørt simuleringen ser vi at maksimalt antall personer inne i heisen på en gang er 10. Samtidig biter vi oss merke i at av de 321 personene som nådde 1. etasje var gjennomsnittstiden 69 tidsenheter, 339 personer kom seg til 2. etasje med en gjennomsnittstid på 52 enheter, mens 326 endte opp i 3. etasje på gjennomsnittlig 71 tidsenheter. Heisen har beveget seg 526 ganger.

Vi ser nærmere på personer som ender opp i 1.etasje og hvordan tiden gjennom systemet på dem endres gjennom kjøringen. Figur 12.5 viser et plott over tid gjennom systemet mot simulasjonsklokken og et histogram over hvor mange personer som brukte hvor lang tid gjennom systemet.

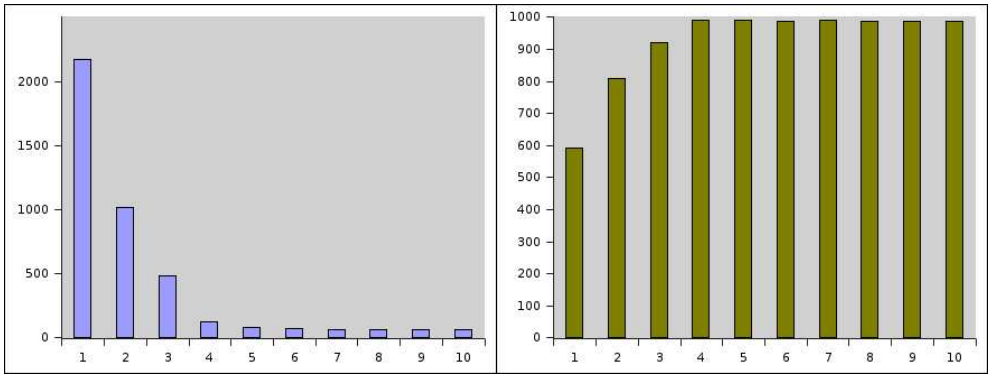


Figur 12.5: Tid gjennom systemet plottet mot systemklokken og histogram over tid brukt til utgangen i 1. etasje.

Dersom vi setter størrelsen på heisen til 6, en ikke uvanlig heisstørrelse, endrer dataene seg som følger:

321 personer nådde 1.etasje på gjennomsnittlig 71 tidsenheter, 339 nådde 2. på gjennomsnittlig 60 enheter, mens 328 nådde 3. på gjennomsnittlig 84 enheter. Heisen beveget seg her bare 498 ganger.

Vi kjører heissimuleringen 10 ganger med heisvognskapasiteter varierende fra 1 til 10 og plotter gjennomsnittstid mot kølengde og antall gjennom systemet mot kølengde i figur 12.6.



Figur 12.6: Gjennomsnittstid og antall personer gjennom systemet plottet mot størrelsen på heisen

Vi ser at allerede med en størrelse på 4 tar heisen med seg de fleste av passasjerene uten at noen trenger å vente for lenge.

Kapittel 13

Konklusjon

13.1 Resultater

Vi har laget en prototype på et system for å bygge købserte simuleringsmodeller i et grafisk grensesnitt som vi har kalt mjmod (*my java modelling*). For å oppnå dette har vi gjort følgende.

- Vi såg først på forskjellige tilnærminger til diskret hendelsessimulering og utviklet en variant av aktivitetsinterrogeringenstilnærmingen som er egnet for å koble opp købaserte modeller grafisk med ikoner og buer. Vi implementerte et rammeverk for simulering i Java, da dette språket innehar en del egenskaper og biblioteker egnet for å jobbe med diskret hendelsessimuleringer.
- Vi bygget opp et språk basert på XML-standarden som vi kalte mjml (*my java modelling language*) for å spesifisere modellene til simuleringsrammeverket vårt. Dette lar oss benytte generelle tekst- eller XML-redigeringsprogram for å sette opp modellene våre. Vi har også i forbindelse med språket laget ett sett av ikoner som egner seg for visuell modellering. Vi har muligheten for å visualisere simuleringsmodellene våre ved å eksportere dem i et generelt filformat for grafer, GraphXML, og studere dem i egnet grafpakke etter eget hjerte.
- Vi laget så et brukergrensesnitt for å laste modeller på mjml-format og gjøre endringer på dem, eller evt. lage modeller fra bunnen av. Dette viser også utvalgte deler av tilstanden til simuleringen under kjøring. Vi kan "klikke oss inn på" hver og en av komponentene som utgjør modellen når som helst og be om en rapport eller endre parametre.

Avslutningsvis viste vi at simuleringspakken vår egner seg til ulike simuleringsoppgaver, som bilvask, datanettverk og en (noe forenklet) heis.

13.1.1 Kommentarer til resultatene

Ved å begrense oss til å se på købasert modellering har vi laget et diskret hendelsessystem der man kan koble opp modeller i et grafisk grensesnitt.

Byggeklossene fungerer

Under bygging av eksempler har funnet ut at byggeklossene vi har definert fungerer bra. Den objektorienterte tilnærmingen vår gjør det lett å legge til nye når det trengs. Samtidig har det vært greit å ikke ha for mange spesialklosser, da det nok vil skapt rot og oversiktlighet.

Mye javakode

Selv om Java, som nevnt i kapittel 3, har en god del egenskaper og redskaper som kjennetegner systemer for å se på diskret hendelsessimulering har vi implementert en god del tusen linjers kode for å få mjmlmod slik som det er.

Det kommer nok delvis av at utviklingen av mjmod har vært av en eksperimentell natur[1] og dermed for eksempel graden av kodegjenbruk ikke har vært optimal.

13.2 Videre arbeid, utvidelser og forbedringer

Underveis har vi sett for oss en del mulige forbedringer og utvidelser til mjmod.

13.2.1 Forbedring av det grafiske grensesnittet for modellering

Den mest nærliggende utvidelsen mjmod er muligheten til å koble opp modellene som en graf i et grafisk grensesnitt.

Modellering i form av en graf

All struktur i simulasjonssystemet vårt legger opp til at vi skal kunne skape kjørbare modeller slik, ikke bare studere de logiske strukturer i modeller. Alle knytnigner mellom objektene er pekere til og fra veldefinerte porter, så det å sette opp et slikt grensesnitt på toppen av vårt system blir en rein programmeringsteknisk oppgave, men tilsynelatende relativt tidkrevende, og derfor utenfor omfanget til denne oppgaven.

Grensesnittet vi ser for oss kan se ut som en hybrid mellom det i Royere og det vi har laget i mjmlModeller. Vi kan tenke oss at vi tar grafmanipulasjonsvinduet til Royere og legger på funksjonaliteten fra mjmlModeller med tanke på modellspesifisering. Flere separate porter på ikonene som representerer aktiviteter med flere inn- og utkanaler vil også være ønskelig for å lette modelloppkoblingen ytterligere, som på figur 9.1.

Mulighet for å stokke om på og slette objekter

Siden rekkefølgen på aktivitetene i en modell i mjmod er det som definerer innbyrdes prioritering, vil det være nyttig å kunne permutere dem i vårt grafiske miljø. Muligheten for å kunne slette ubrukte objekter vil også kunne gagne en modelleringsprosess. Slike endringer må per dags dato gjøres i mjml-filen, men vi kan utvide det grafiske grensesnittet for modellering til å støtte også dette.

[1] Til tross for at systemet som foreligger er vår andre prototype av mjmod - versjon 0.1.

13.2.2 Grafiske tilbakemeldinger

Vi kan også utvide settet av grafiske tilbakemeldinger ved å lage mer spesifiserte monitor-funksjoner for de ulike objektene. For eksempel kan vi vise *hvilke* enheter som er i en aktivitet eller kø. Dette blir også en rein programmeringsteknisk oppgave, da vi har vist i kapittel 8 at dette generelt lar seg gjøre.

Et alternativ for å skape animasjoner er å benytte GraphXML sin animasjonsfunksjonalitet, enten underveis i simuleringen, eller ved hjelp av postprosessering av statistiske data.

13.2.3 Økt lesbarhet ved integrerte komponenter og ikoner

Ved å koble flere basiskomponenter sammen til ett grafisk element kan vi øke lesbarheten til modellen. Dette kan feks. være en generator som tagger enhetene den genererer (fra heiseksempelet). Vi kan også tenke oss å lage aktiviteter som inkorporerer køene de lytter på, dersom de lytter ekslusivt på disse. (Utslagsvasken er implementert slik nå.) Dessuten kan vi lage spesialkomponenter for visse oppgaver, som feks. heissjakten i heisen. Dette kan inngå i et fremtidig bibliotek myntet å modellere med mjml.

13.2.4 Ekstra komponenter for modellering/ekstra funksjonalitet i eksisterende komponenter

Det å legge for mye funksjonalitet inn i disse basiskomponentene eller å opprette for mange spesifiserte komponenter kan gjøre modeller for uoversiktelige. En del ekstra funksjonalitet kan likevel tenkes implementert.

Vekter/størrelser på enheter

I stedet for at alle enheter er like "store" i kø og aktivitetssammenheng, som nå, har vi klargjort et system for å vekte dem ulikt og dermed feks. trenge ulik grad av tid/aktiviteter/plass. Dette kan nå løses ved hjelp av en taggleser og aktiviteter som trenger flere statiske enheter, som kanskje ikke er den mest oversiktlige løsningen.

Å bygge opp enheter av flere enheter

En enhet kan tenkes å være sammensatt av flere enheter. Dette kan være nyttig dersom vi feks. har en pøl av lastebiler og en annen pøl av ulike tilhengere til lastebiler. Da kan vi, etterhvert som det trengs, skape forskjellige "lastebil med tilhenger"-objekter som skal henge sammen med hverandre gjennom flere aktiviteter i modellen, før de deles opp igjen til lastebil og tilhenger og plasseres tilbake i sine respektive køer, atter klar for bruk.

Boolsk logikk på innkøene

Kanskje kan en del flere problemstillinger kan modelleres dersom vi kan spesifisere betingelser for aktivitetene ved boolsk logikk. I stedet for som nå, at *ressurser* skal ha enheter til alle innkøene sine og *blandere* skal ha en og bare en, kan vi tenke oss at vi kan spesifisere at aktivitetens betingelser er oppfylt for

eksempel dersom det er enheter på inngang 1 og enten 2 eller 3. Her ser vi for oss at bruk av hele spekteret av boolske operatorer; OR, AND, XOR og NOT.

13.2.5 Etterbehandling av statistiske data

Til nå skriver vi all statistisk informasjon til filer for etterbehandling i . Vi kan tenke oss et system der vi automatisk tegner grafer og diagrammer av statistikk for utvalgte data etter kjøring. Til det er det laget flere biblioteker til Java - blant annet i det frie UJAC (*Useful JAVA Application Components*).[2]

13.2.6 XML-relaterte forbedringer

Ved å bruke SAX-bibliotekene til Java (for tolking av XML-filer) istedet for vår egenimplementerte tolker vil vi kunne gjøre mjml-filsystemet vår mer rigid. Vi følger nemlig ikke XML-spesifikasjonene på et punkt; streng syntaks. En XML-parser skal egentlig abortere dersom feks. en slutt-tagg ikke samsvarer med en start-tagg.[3] Vi avbryter ikke en tolking av en modell selv om dette skjer, men prøver å fortsette og gir bare en feilmelding til konsollet.

13.2.7 Tolke mjml til modeller for andre simulatorer

Nå når vi har vist at byggeklossene i mjml kan brukes til å modellere købaserte systemer kan vi tenke oss at vi vil implementere oversettere fra mjml til andre simulatorer som baserer seg på aktivitetsinterrogering. Dette vil innebære å bygge basisenhetene i modelleringsspråket til gitte simulator, for så å tilpasse mjml-tolkeren.

13.3 Begrensninger i måten vi modellerer på

Hvilke begrensninger ligger i måten vi modellerer problemer på?

13.3.1 Problemer med spesifikke modeller

Vi kan bare benytte systemet vårt til å modellere problemer som kan reduseres til købasert modellering. Dersom en modell blir for komplisert å spesifisere vil vi ikke kunne uttrykke den i vårt modelleringsspråk. Det vil som nevnt tidligere alltid være en avveining mellom generalitet og fleksibilitet på den ene siden og oversiktlighet og enkelhet på den andre siden involvert i en modelleringssystem-sutvikling. Vi har lagt oss på et temmelig høyt abstraksjonsnivå for utvikling av modeller og har selvsagt måtte betale for det i form av fleksibilitet.

[2] http://ujac.sourceforge.net

[3] Grunnen til at W3C har vært så streng med at man skal avbryte parsingen og ikke "gjette seg frem" dersom koden ikke er innen for standarden er nok for å hindre at selskaper skal gjøre det samme med XML som Microsoft har gjort med HTML. Microsoft prøver med sin Internett Explorer som kjent å sette en "standard" for gjetting i HTML-dokumenter. Ved sitt tilnærmede monopol på markedet for nettleserer prøver de på den måten å kuppe HTML-standarden og sikre seg et konstant forsprang på sine konkurrenter.

13.3.2 Problemer med store modeller

Systemet vårt har problemer med store modeller av to grunner; uoversiktelighet og minnebruk.

Minnebruk

I store modeller blir det gjerne veldig mange enheter i systemet under kjøring. Siden hver av dem tar vare på en hel del statistikk om sin gang gjennom systemet opptar de nødvendigvis en del minne.

Inspirert av Helsgaun har vi dermed implementert et system som kaster enheter som er ferdig i systemet til java sin søppeltømmer etter at de er blitt loppet for statistikk om deres ferd gjennom modellen. Dermed blir det ikke flere enheter i systemet enn de som er aktive eller befinner seg i en eller annen kø.

Uoversiktleighet

Dersom grafene til systemet vårt blir for store kan man miste oversikten over den strukturelle logikken i modellen. Dette kan løses ved å bruke gode visualiseringsverktøy for grafer. Royere, som vi har brukt i våre eksempler, har en del funksjoner for å få oversikt over en komplisert graf. For eksempel kan man anvende denne fiskeøye-funksjonen nevnt tidligere, eller man kan la det finne strukturelle klynger i grafstrukturen. Man kan også rett og slett slå av noe informasjon som er knyttet til grafen, som feks. nodenavn, og oppnå en mer lesbar graf.

13.3.3 Mangler ved Royere

Navn på kanter

Dessverre viser ikke Royere navn på kanter. Derfor får vi ikke utnyttet GraphXML til sitt fulle for modellvisualisering. Vi løser dette ved å gi ulik farge på ulike kanter i grafen. Et annet ankepunkt er at Royere tegner en dobbeltpil, dersom vi har en kant fra en node til en annen og vise versa. Dette er jo vanlig i klassisk grafbehandling, men vi skulle gjerne ha fått det til å se anderledes ut.

Porter

Dette bringer oss til en annen svakhet ved Royere/GraphXML, nemlig mangelen på å "porter" på nodene. Dette ville vært ønskelig under visualisering (og i enda større grad under modellering) av våre modeller, men dette er nok en svakhet ved å bruke generelle grafprogrammer fremfor et spesialutviklet verktøy for vårt formål.

Blanding av rettede og urettede kanter

Et annet problem med Royere er at det ikke støtter grafer som veksler mellom rettede og urettede kanter. Dermed får vi piler på kantene fra fordelingene til aktiviteter og prosesser samt fra blåpapirsenheter til generatorer, noe som kan gi uheldige assosiasjoner. Disse kantene forteller nemlig ikke noe om enhetsflyten i modellen, slik som samtlige andre kanter, men snarere noe om statistiske og generelle sammenhenger.

13.3.4 Mangler ved Java

Siden Java 2 versjon 1.5 skal lanseres med brask og bram om ikke lenge er det på sin plass med litt kommentarer til versjon 1.4.

Nødvendigheten av typetildeling under kjøring

Vi har implementert systemet vårt i versjon 1.4 av Java 2. Den største mangelen i dette språket, i forbindelse med vårt arbeide, er at den ikke støtter generiske typer, slik som feks. C++[4] gjør. Det innebærer at vi ikke kan spesifisere hvilke *typer* objekter en samling (*Collection*) består av. Dette gjør at vi selv må huske på hvilke objekter vi putter i en samling under koding. Under kjøring må vi gjøre sanntids typetildeling (*runtime casting*). Dette kan føre med seg feil som er vanskelig å spore opp.

Vi kan eksempelvis definere en kø av enheter i mjmod slik:

```
Que bilkø = new FifoQueue("Bilkø");
```

For å hente ut bil nummer n som en enhet fra den må vi kalle:

```
Entity enBil = (entity) bilkø.get(n);
```

Dette kan gi kjøretidsfeilmeldinger om typefeil om vi ikke husker på å bare legge enheter i køen. Vi kan omgå problemet med å kapsle hvert oppslag i listen inn i prøv- og grip-klausuler (*try-catch*), men dette er ikke særlig hensiktsmessig i alle tilfeller.

Den kommende Java2 1.5

Generiske typer Heldigvis har Sun, med introduksjonen av Java2 1.5, blant annet innført generiske typer.[5] Dermed kan vi spesifisere at en liste består av strenger på følgende sett:

```
<String> ord = new LinkedList<String>();
```

Og vi kan dermed hente ut et ord fra listen slik:

```
String etOrd = ord.get(n);
```

Dette gjør altså at vi kan finne typefeil relatert til samlinger under kompilering, og vi kan redusere eventuelle kjøretidsfeil.[6]

Forbedrede for-løkker En annen grei utvidelse er en forbedring av for-løkker. Dette er mulig nettopp på grunn av innføringen av generiske typer. Dersom vi har definert listen med ord som tidligere kan vi kort og konsist implementere en iterator til listen slik:

```
for (String streng : ord) { ... }
```

[4] *Bjarne Stroustrup's FAQ,* http://www.research.att.com/~bs/bs_faq.html

[5] Mughal, K A, *Java 1.5 - One Tenth Up (Advanced Tpoics in Java)*, Teknisk rapport, Institutt for informatikk, Universitetet i Bergen, 2004

Austin, C, *J2SE 1.5 in a Nutshell*, http://java.sun.com/developer/technicalArticles/releases/j2se15/

[6] Dette gjør Java til et, i enda større grad, *typet* språk, men Standard ML (http://www.smlnj.org/) er enda kongen på haugen innen typing.

Statisk import Den tredje nyervervelsen i Java er muligheten for statisk import. Det vil si at vi kan benytte oss av funksjoner og variabler fra andre klasser uten å arve fra dem. Dette kan man se på som et steg på veien mot multippel arv som mange har etterspurt.

Dessverre er ikke Java 1.5 offisielt lansert enda[7] så vi har ikke benyttet oss av disse egenskapene.

[7]Det finnes betaversjoner tilgjengelig for de eventyrlystne, dog.

For mer informasjon om mjmod se http://www.ii.uib.no/~mortene/mjmod/.

Bibliografi

[1] Abrief, L B og Speir, N A, *A UML Tool for an Automatic Generation of Simulation Programs,* Teknisk rapport, Department of Computing Science, University of Newcastle Upon Tyne, Newcastle upon Tyne, England, 1999

[2] Archer, N P, Ma, R og Chen, S, *A basis for an object-oriented discrete simulation library*, Information and Decision Technologies, volume 19, side 455-469, 1993

[3] Au, G og Paul, R J, *A Graphical Discrete Event Simulation Environment*, INFOR, volume 35 nummer 2, side 121-137, 1997

[4] Au, G og Paul, R J, *Visual interactive modelling: A pictoral simulation specification system,* European Journal of Operational Research, nummer 91, side 14-26, 1996

[5] Banks, J, *Handbook of Simulation - Principles, Methodology, Advances, Aplications, and Practice*, Wiley-Interscience / John Wiley & Sons, Inc, New York, USA, 1998

[6] Bergmann, Ø, *Discrete event simulation in Java with applications in rodent navigation,* Hovedfagsoppgave, Institutt for Informatikk og avdeling for fysiologi, Universitetet i Bergen, Bergen, 2002

[7] Birthwistle, G M, Dahl, O-J, Myrhaug, B, Nygaard, K, *Simula Begin*, AUERBACH Publishers Inc, Philadelphia, Pa, 1973

[8] Dahl, O-J, *Discrete Event Simulation Languages*, Norwegian Computing Center, Oslo, 196?

[9] Fishman, G S, *Discrete-Event Simulation*, Springer Verlag, New York, USA, 2001

[10] Flanagan, D, *Java Foundation Classes in a Nutshell*, O'Reilly & Associates, Inc. , Sebastopol, California, USA, 1999

[11] Flanagan, D, *Java in a Nutshell*, O'Reilly & Associates, Inc. , Sebastopol, California, USA, 2002

[12] Garrido, J M, *Object-Oriented Discrete-Event Simulation with Java,* Kluwer Academic/Plenum Publishers, New York, 2002

[13] Helsgaun, K, *Discrete Event Simulation in Java,* Teknisk rapport, Department of Computer Science, Roskilde University, DK-4000 Roskilde, Denmark, 2000, http://www.dat.ruc.dk/~keld/research/JAVASIMULATION/JAVASIMULATION-1.1/docs/Report.pdf

[14] Herman, I og Marshall, M S, *An object-oriented approach to graph visualization*, Teknisk rapport, CWI, Amsterdam, Nederland, 2001

[15] Hunter, D, Cagle, K, Dix, C, Kovack,R, Pinnock, J, Rafter, J, *XML, 2.utgave*, IDG Norge Books, Oslo, 2002

[16] Joines, J A og Roberts, S D, *Design of Object-Oriented Simulations in C++*, Teknisk rapport, Department of Industrial Engineering, North Carolina State University, Raleigh NC 27695-7906, USA, 1996

[17] Knuth, D E, 1968, *Fundamental Algorithms,* volume 1 av *The Art of Computer Programming,*Addison-Wesley

[18] McNab, R og Howell, F W, *A discrete event simualtion package for Java - with applications in computer system modelling*, Teknisk rapport, Department of Computer Science, The University of Edinburgh, Edinburgh, Strorbritannia, 1997

[19] McNab, R og Howell, F W, *Using Java for Discrete Event Simulation*, Teknisk rapport, Department of Computer Science, The University of Edinburgh, Edinburgh, Strorbritannia, 1996

[20] Nance, R E, *A History of Discrete Event Simulation Programming Languages*, ACM SIGPLAN Notices, volum 28, nummer 3, mars 1993

[21] Paul, R J, Odhabi, H I og Macredie, R D, *The Four Phase Method for Modelling Complex Systems*, Proceedings of the Winter Simulation Conference ed. Andradóttir, S, Healy, K J, Withers, D H, Nelson, B L, 1997

[22] Paul, R J, Odhabi, H I og Macredie, R D, *Making Simulation More Accessible in Manufacturing Systems through a 'Four Phase' Approach*, PRoceedings of the Winter Simulation Conference ed. Medeiros, D J, Watson, E F, Carson, J S, Manivannan, M S, 1998

[23] Paul, R J, Odhabi, H I og Macredie, R D, *Java Iconic Visual Environment for Simualtion (JIVESim)*, Computers & Industrial Engineering, volume 37, side 243-246, 1999

[24] Pidd, M, *Object Orientation, Discrete Simulation and the Three-Phase Approach,* Journal of the Operational Research Society, volum 46, side 362-374, 1995

[25] Prähofer, H, Sametinger, J og Stritzinger, A, *Discrete Event Simulation Using the JavaBeans Component Model,* Teknisk rapport, Department of Systems Theory and Information Engineering og C. Doppler Laboratory for Software Engineering, Johannes Kepler University, A-4040 Linz, Østerrike, 1999

[26] Prähofer, H, Sametinger, J og Stritzinger, A, *Component Frameworks: A Case Study*, Proceedings of TOOLS 30 conference, Santa Barbara, USA, IEEE Society Press, 1999

[27] Prähofer, H, Sametinger, J og Stritzinger, A, *Consepts and Architecture of a Simulation Framework based on JavaBeans,* FGCS, Elsevier Publishers, 2000

[28] Prähofer, H, Sametinger, J og Stritzinger, A, *Building Reusable Simulation Components*, Proceedings of WEBSIM2000, San Diego, 2000

[29] Prähofer, H, Sametinger, J og Stritzinger, A, *Using JavaBeans to teach Simulation and using Simulation to teach Java*, Proceedings of ESM98, Den tolvte europeiske multikonferansen om simulering, Manchester, Storbritannia, 16.-19. juni, 1998

[30] Sethi, R, *Programming Languages : Concepts & Constructs,* Andre utgave, Addison-Wesley, USA, 1997

[31] Stritzinger, A, *A Component-Based Modelling Approach*, Proceedings of the WOON'96, St. Petersburg, Russland, 20-21 juni, 1996

[32] Swain, J J, *Simulation (Software) Surveys*, OR/MR Today, Oktober 1997, 1999, Februar 2001

[33] Tyan, H-Y og Hou, C-J, *JavaSim: A Component-Based Compositional Network Simulation Environment*, Teknisk rapport, Department of Electrical Engineering, The Ohio State University, Columbus, USA, 1997

[34] Wilensky, U, *NetLogo*, http://ccl.northwestern.edu/netlogo. Center for Connected Learning and Computer-Based Modeling. Northwestern University, Evanston, IL, 1999

[35] Zeigler, B P, *Object-Oriented Modelling and Discrete-Event Simulation*, Advances in Computers, volum 33, side 67-114, 1991

[36] IBM, *Unified Modelling Language Resource Center,* http://www.rational.com/uml/index.jsp

[37] McNab, R, *A Guide to the simjava package,* http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.2/doc/simjava_guide/

[38] W3C, *Extensible Markup Language (XML)*, http://www.w3.org/XML

(Alle pekere til sider på verdensveven er á jour per 11. juni 2004.)

Tillegg A

Redskaper og biblioteker brukt

I dette appendikset vil jeg gi litt flere detaljer om hvilke redskaper som har blitt benyttet under arbeide med denne oppgaven, og takke dem som står bak.

A.1 javasimulation

Helsgaun[13] sin pakke med biblioteker for simulering i Java, javasimulation, har vært til stor hjelp under utformingen av en ny simuleringspakke. I dette systemet implementerte vi bla en heissimulering som spesifisert av Knuth for å studere diskret hendelsessimulering for å få oversikt over denne typen simulering. Det at den går gjennom de tre hovedverdenssynene innen dette fagfeltet gav dyp innsikt i hvilke forskjeller som egentlig finnes på dem.

Vi har også (med tillatelse) brukt Helsgaun sin javakode for å generere stokastiske data.

A.2 GraphXML og Royere

Dette har jeg vært inne på i selve oppgaveteksten, men jeg vil gjerne rette en stor takk til dem på CWI som står bak dette verktøyet.

A.3 Gorilla, SVG og Sodipodi

For å lage ikonene, som vi har benyttet både i det grafiske språket for modellering og det grafiske grensesnittet, tok vi utgangspunkt i et GNU/GPL-lisensert[1]

[1] I en fotnote kan vi, som så ofte innen datavitenskapen, trekke linjer til Donald Knuth. Han er et ikon for GNU-bevegelsen. Han har økt forståelsen for at datakode kan sees på som en kunstform i sin bokserie *The Art of Computer Programming*. For at folk skal få sette pris på kunsten må den jo sees, ergo kildene må være åpne. Om åpen kildekode sier han: *"I'm more than sympathetic to that; I consider it absolutely indespensable for progress."* (Sitat fra *Rebel Code*, av Glyn Moody.) Han står også bak et viktig prosjekt basert på åpen kildekode; TEX.

ikonsett kalt Gorilla. Det er utviklet av Jakub Steiner[2] og spesifisert i filformatet SVG - skalerbar vektorgrafikk *(Scalable Vector Graphics)*. Dette er et XML-basert filformat som er sertifisert av W3C[3] og begynner å bli standard for å spesifisere blant annet ikoner for Linux-skrivebordene KDE[4] og GNOME[5]. Adobe har også trykket formatet til sitt bryst.[6]

SVG-formatet lot oss redigere og klippe og lime etter eget hjerte, med programmet Sodipodi[7]. Dette er et fritt program utviklet for nettopp redigering av SVG-dokumenter.

[2] http://librsvg.sourceforge.net/theme.php og http://jimmac.musichall.cz/gorilla.php3 og
[3] http://www.w3.org/Graphics/SVG/
[4] http://www.kde.org
[5] http://www.gnome.org
[6] http://www.adobe.com/svg/
[7] http://www.sodipodi.com/

Tillegg B

Å skrive norsk

> *"The basic tool for the manipulation of reality is the manipulation of words. If you can control the meaning of words, you can control the people who must use the words."*
> Philip K. Dick

B.1 Hvorfor skrive norsk?

Valget av å skrive oppgaven på norsk viste seg å medføre en større utfordring i seg selv enn først anslått. Det aktive dataterminologiske ordforrådet oversvømmes av engelske ord.[1] Stort sett alt av pensum, bøker og artikler jeg har lest i forbindelse med informatikkstudiene har vært på engelsk.[2] Dette har selvsagt satt sine spor. Jeg har prøvd så godt jeg har kunnet å leite opp eksisterende oversettelser av engelske termer. I tilfeller der jeg vil understreke hvilket engelsk ord som er oversatt oppgir jeg dette i parenteser.

B.2 Behov for en standard

Norsk språkråd har en artikkel ute på verdensveven kalt *Norske avløysarord*[3], som bla sier:

> "Fagspråket er ein arbeidsreiskap som stiller spesielle krav, ikkje minst til eintydigheit og standardisering. Derfor blir det lagt vekt på presise definisjonar. Fagterminologien er ofte utvikla i systematisk samarbeid i spesialistgrupper, f.eks. ved at det blir laga fagordbøker."

Språrådet har la da også frem *Handlingsplan for norsk språk og IKT - Revidert utgave*[4] i juni 2001 der vi spesielt biter oss merke i punkt 6 på målsetningen:

[1] Selv i norsk dagligtale florerer ord som *teambuilding, theme* (i skrivebords-verden), *whiplash, layout og catwalk,* selv om vi har fullgode, om ikke bedre, alternativer til dem *lagutvikling, ham/tema, nakkesleng, bladbunad og motemolo.*

[2] Faktisk har vel alt datarelatert lektyre siden "*BASIC for Commodore64'en*" i min spede barndom vært på engelsk.

[3] http://www.sprakrad.no/hsavlos.htm

[4] Tilgjengelig på: http://www.sprakrad.no/iktrev.htm

> "En standardisert norsk IKT-terminologi må utvikles og stilles til rådighet"

En løsning planen forespeiler på dette er å opprette en språkbank for IKT. De eneste ordbøkene jeg har funnet fra språkrådet som går på IKT *Norsk dataordbok*[5], *Ei Lita dataordliste for nettet*[6] og tildels *På godt norsk*[7].

Jeg er enig med handlingsplanen til språkrådet i at vi trenger en språkbank for IKT. Ikke bare med tanke på (vitenskapelige) artikler men også for dataprogrammer generelt. En artikkel er greiere å forholde seg til om vi slepper å referere til engelske ord stadig vekk. Ethvert grensesnitt vil være mye mer brukervennlig om man kjenner igjen ordlyden i kommandoer og tilbakemeldinger fra program til program (uten at disse trenger å være fra samme produsent). Dette er da også grunnen til at mange slett ikke vil ha et norsk skrivebord på datamaskinen sin.

B.3 Eksisterende ordbøker og leksika

Siste versjon av Språkrådets *Norsk dataordbok* kom i 1993. Den har jeg benyttet meg av så langt den rakk. Mye har hendt siden 1993. Termer som for eksempel fremdrifts-bjelke *(progress bar)*, eksisterte ikke da.[8]

Hjelpemidler som PCWorld sitt *Dataleksikon*[9] er mer forklaringer på - ikke så mye oversettelser av - begreper knyttet til IKT.

Skolenettet har et ordbok kalt IKT-ordboka[10]. Denne definerer en del begreper på norsk. For eksempel har de en veldig bra oversettelse på CSS (Cascading Style Sheets); Gjennomgripende stilark, men de går over bekken etter vann og bruker det engelske ordet for blåtann - *bluetooth*.

B.4 Et eksempel; XML

Noen ganger har jeg ikke funnet fullgode fornorsking av uttrykk og prøvd å innføre logiske oversettelser. Et eksempel er *extensible markup language* som vanligvis bare er betegnet ved forkortelsen XML. Dataleksikonet til PCWorld Norge, for eksempel, definerer XML slik:

> "XML: eXtensible Markup Language. Et språk for lagring og utveksling av strukturerte data via f.eks. Internett eller intranett. Språket ligner på HTML, som benyttes for å beskrive Web-sider, mens det viktigste bruksområdet for XML er datautveksling. XML er i motsetning til HTML ikke et fastlåst språk, men utvidbart (derav "extensible")."

[5] Hofstad, K, Løland, S og Scott, P, *Norsk dataordbok,* Universitetsforlaget, Oslo, 1993

[6] http://www.sprakrad.no/dataord.htm

[7] http://www.sprakrad.no/paanorsk.htm

[8] Noe som kanskje kan underbygge IBM sitt omstridte patent på slike fremgangsindikatorer fra 1990 (http://swpat.ffii.org/patents/txt/ep/0394/160/). På den annen side synes jeg å huske at dataspill som tok tid å starte på 80-tallet var ivrig med å vise at noe faktisk hendte, så begreper som "kjent teknikk" (*prior art*) må vel ugyldiggjøre denne patenten så vi slepper å betale lisenser til IBM for GUI'en vår. Faktisk bryter vi også et patent på *quicksort*. Men; selv om EU i disse dager tilsynelatende velger en fæl patentlovgiving, er vel dette hverken tid eller sted for å gå i detaljer rundt dette.

[9] http://www.pcworld.no/dataleksikon/

[10] http://skolenettet.ls.no/imaker?id=43249&method=list_ord&malgruppe=0&trinn=0&alfa=A

ML delen har jeg sett "oversatt" som *markupspråk* (Nordlund, B, *XML og mobilt internett,* Norsk regnesentral).

Jeg valgte å oversette XML med "utvidbart[11] formateringsspråk[12]" noe jeg mener er dekkende da det er nettopp muligheten for utvidelser av dette metaspråket som skiller det fra språk som HTML.

> *"Translation is the art of failure."*
> - Umberto Eco

[11] Ifølge http://www.clue.no: adj. strekkbar, tøyelig, utvidbar adj. uttrekkbar (extensible language) [edb] ekstensivt språk

[12] Ifølge http://www.clue.no: (markup language) [edb] formateringsspråk

Tillegg C

Utvalgt java-kode

C.1 SimStep() - hjertet av simulatoren

```
public double simStep(double maxTime) {
    while (!finished) {
     //Phase 1 - check the conditions
      int numberOfActivities = activityList.size();
      for (int x = 0; x < numberOfActivities; x++) {
        Activity act;
        act = (Activity) activityList.get(x);
        if (act.conditions(simTime)) {
          timeQueue.in(act, simTime);
        }
      }
     //If the timeQueue is empty at this point - finish the simulation
      if (timeQueue.isEmpty()) {
        finished = true;
      }
      else {
        //Phase 2 - uptate simTime
        Proc nextUp = (Proc) timeQueue.out(simTime);
        simTime = nextUp.getTimeScheduled();
        //Phase 3 - run nextUp
        //If we've run out of simTime, only the conditional processes are executed
        //so that the entities generated before maxTime are served
        if (simTime >= maxTime) {
          if (stopUnconditionallyOnMaxtime) {
            finished=true;
          }
          else {
            if (progressBar!=null) setIndeterminate(true);
            try {
              Activity a = (Activity) nextUp;
              double hold = nextUp.actions(simTime);
              if (hold >= 0) {
```

```
            timeQueue.in(nextUp, simTime);
          }
        }
        //If an element in the timeQueue is not an Activity - skip it
        catch (Exception ClassCastException) {
          System.out.println("Unconditional process " + nextUp.name +
                             " stopped at " + simTime);
        }
      }
    }
    //Otherwise we have more time to spend and everything is good...
    else {
      //timeQueue.remove(nextUp);
      double hold = nextUp.actions(simTime);
      if (hold >= 0) {
        timeQueue.in(nextUp, simTime);
      }
    }
  }
 }
  return simTime;
}
```

Tillegg D

mjml - utdypet

D.1 Generelle parametre til modellen

Først i mjml-filen har vi et hode som spesifiserer overordnede data om modellen:

- Navn: <name>
- Hvor lang tid den skal kjøre (i simulasjonstid): <time>
- Om simuleringen skal stoppe når den når denne tiden, eller om den bare skal avbryte de aktivitetene som er satt til å måtte skje og fortsette til alle de kondisjonale også er ferdige: <stop-unconditionaly-on-maxtime>
- Om simuleringen skal gå fil alle dynamiske enheter er ute av systemet, selv om aktiviteter har betingelser oppfylt: <stop-on-empty>
- Generell info om modellen: <info>

D.2 Argumenter til de ulike byggeklossene

På tilsvarende måte spesifiseres de andre komponentene som skal inngå i modellen vår.

D.2.1 Fordelinger

Fordelinger trenger følgende argumenter:

- Nummer: <no>
- Entydig navn: <navn>
- Type, kan være en av følgende; "uniform", "normal", "negexp", "poisson", "constant", "integer" eller fil.

Alle fordelingstypene utenom *file* og *integer* krever dessuten:

- Middelverdi: <mean>

File trenger:

- Filnavn: <file>

For å definere verdimengden til fordelinger av typen *integer* trenger:

- Minste integerverdi: <min>
- Største integerverdi:<max>

Alternativt kan man også sette (på alle typer utenom *file*):

- initialverdi: <seed>
- Antall verdier som skal returneres i løpet av simuleringen: <number-of-values-to-generate>

D.2.2 Enheter

Enhetene definert i mjml-filen er blåpapirenhetene som skal inngå i modellen. De tar følgende argumenter:

- Nummer: <no>
- Entydig navn: <name>
- Type, kan være enten "dynamic" eller "static": <type>

D.2.3 Køer

Køene tar følgende argumenter:

- Nummer: <no>
- Entydig navn: <name>
- Maksimumsstørrelse, 0 for uendelig: <maxsize>

Utslagsvasker defineres som en kø og kan ta følgende ekstra-argument:

- Filnavn for logging av statistikk: <logfile>

D.2.4 Prosesser

Generatorer

Generatorer tar følgende argumenter:

- Nummer: <no>
- Entydig navn: <name>
- Navn til blåpapirsenhet: <blueprint>
- Nummer på blåpapirsenhet: <blueprint-pointer>
- Antall enheter som skal genereres, 0 for uendelig: <generate>
- Navn til fordeling: <distribution>

- Nummer på fordeling: <distribution-pointer>
- Navn på kø enheter skal leveres til: <outqueue>
- Nummer til kø enheter skal leveres til: <outqueue-pointer>

Ekstra-argument:

- Sprett dersom leveranse ikke er mulig, *"true"* eller *"false":* <bounce-on-no-delivery>

D.2.5 Aktiviteter

Aktiviteter tar følgende argumenter:

- Nummer: <no>
- Entydig navn: <name>
- Type aktivitet, "resource", "tagwriter", "tagreader", "merger", "splitter" eller "priorityresource": <type>
- Navn til fordeling: <distribution>
- Nummer på fordeling: <distribution-pointer>
- Liste over køer enheter skal leveres til: <outqueues>
- Liste over køer enheter skal hentes fra: <inqueues>

Dersom vi definerer en merger må den ha:

- Navn til blåpapirsenhet: <blueprint>
- Nummer på blåpapirsenhet: <blueprint-pointer>

Ekstra-argumenter:

- Sprett dersom leveranse ikke er mulig, *"true"* eller *"false":* <bounce-on-no-delivery>
- Hvor lenge skal aktiviteten vente dersom den ikke klarer å levere, før den returnerer enheter den har tatt opp til køene de kom fra: <wait-on-no-delivery>
- Navn på taggen enheter som går gjennom skal få: <set-tag-name>
- Verdi på taggen enheter som går gjennom skal få: <set-tag-value>

Elementene i listen med køer spesifiseres slik:

- Navn: <name>
- Nummer: <pointer>

D.3 dtd-beskrivelse av mjml

```
<?xml version='1.0' encoding="UTF-8"?>
<!ELEMENT mjml
   (head,simulation)
>
<!ELEMENT head
   (name,time,(stop-unconditionaly-on-maxtime?,stop-when-empty,info)?)
>
<!ELEMENT simulation
   (imports,distributions,entities,queues,processes,activities)
>
<!ELEMENT blueprint
   (#PCDATA)
>
<!ELEMENT blueprint-pointer
   (#PCDATA)
>
<!ELEMENT bounce-on-no-delivery
   (#PCDATA)
>
<!ELEMENT distribution
   (no,type,name,mean,(min,max)?,number-of-values-to-generate?)*
>
<!ELEMENT distribution-pointer
   (#PCDATA)
>
<!ELEMENT number-of-values-to-generate
   (#PCDATA)
>
<!ELEMENT distributions
   (distribution+)
>
<!ELEMENT entities
   (entity+)
>
<!ELEMENT entity
   (no,name,type,weight?)
>
<!ELEMENT expiration-distribution
   (#PCDATA)
>
<!ELEMENT expiration-distribution-pointer
   (#PCDATA)
>
<!ELEMENT generate
   (#PCDATA)
>
<!ELEMENT generator
   (no,name,blueprint,blueprint-pointer,generate,distribution,distribution-pointer,(e
```

```
>
<!ELEMENT processes
   (generator+)
>
<!ELEMENT imports
   EMPTY
>
<!ELEMENT info
   (#PCDATA)
>
<!ELEMENT inqueues
   (queue+)
>
<!ELEMENT logfile
   (#PCDATA)
>
<!ELEMENT look-for-tag-name
   (#PCDATA)
>
<!ELEMENT look-for-tag-value
   (#PCDATA)
>
<!ELEMENT max
   (#PCDATA)
>
<!ELEMENT maxsize
   (#PCDATA)
>
<!ELEMENT mean
   (#PCDATA)
>
<!ELEMENT min
   (#PCDATA)
>
<!ELEMENT name
   (#PCDATA)
>
<!ELEMENT no
   (#PCDATA)
>
<!ELEMENT outqueue
   (#PCDATA)
>
<!ELEMENT outqueue-pointer
   (#PCDATA)
>
<!ELEMENT outqueues
   (queue+)
>
<!ELEMENT pointer
```

```
   (#PCDATA)
>
<!ELEMENT queue
   (((no,type,name,(maxsize|logfile))|(name,pointer)))
>
<!ELEMENT queues
   (queue+)
>
<!ELEMENT read-queue
   (queue)
>
<!ELEMENT activity
   (no,name,type,distribution,distribution-pointer,(((set-tag-name|(blueprint,generat
>
<!ELEMENT activities
   (resource+)
>
<!ELEMENT set-tag-name
   (#PCDATA)
>
<!ELEMENT stop-unconditionaly-on-maxtime
   (#PCDATA)
>
<!ELEMENT stop-when-empty
   (#PCDATA)
>
<!ELEMENT time
   (#PCDATA)
>
<!ELEMENT type
   (#PCDATA)
>
<!ELEMENT wait-on-no-delivery
   (#PCDATA)
>
<!ELEMENT watch-empty
   (#PCDATA)
>
<!ELEMENT weight
   (#PCDATA)
>
<!ELEMENT yes-queue
   (queue)
>
```

D.4 Utvalgt mjml-kode

D.4.1 Bilvasken

```
<?xml version="1.0"?>
<!DOCTYPE mjml SYSTEM "file:mjml.dtd">
<mjml>
<head>
        <name>Bilvask</name>
        <time>100000</time>
        <info>Bilvask med én vaskemaskin</info>
</head>
<simulation>
        <imports>
        </imports>

        <distributions>
                <distribution>
                        <no>1</no>
                        <name>Generisk</name>
                        <type>negexp</type>
                        <mean>110</mean>
                </distribution>
                <distribution>
                        <no>2</no>
                        <name>Vasketid</name>
                        <type>constant</type>
                        <mean>10</mean>
                </distribution>
                <distribution>
                        <no>3</no>
                        <name>Instant</name>
                        <type>constant</type>
                        <mean>0</mean>
                </distribution>
                <distribution>
                        <no>4</no>
                        <name>Fil</name>
                        <type>file</type>
                        <file>/home/johs/src/mojomod/times.txt</file>
                </distribution>
        </distributions>

        <entities>
                <entity>
                        <no>1</no>
                        <name>Skitten bil</name>
                        <type>dynamic</type>
                </entity>
                <entity>
```

```
                <no>2</no>
                <name>Vaskemann</name>
                <type>static</type>
        </entity>
</entities>

<queues>
        <queue>
                <no>1</no>
                <type>FifoQueue</type>
                <name>Bilkø</name>
                <maxsize>0<maxsize>
        </queue>
        <queue>
                <no>2</no>
                <type>LifoQueue</type>
                <name>Pauserom</name>
                <maxsize>0<maxsize>
        </queue>
        <queue>
                <no>3</no>
                <type>Sink</type>
                <name>Utkjørsel</name>
            <logfile>/home/johs/src/mojomod/bilvask1-sink.cvs</logfile>
        </queue>
</queues>

<processes>
        <generator>
                <no>1</no>
                <name>Innkjørsel</name>
                <blueprint>Skitten bil<blueprint>
                <blueprint-pointer>1</blueprint-pointer>
                <generate>0</generate>
                <distribution>Generisk</distribution>
                <distribution-pointer>1</distribution-pointer>
                <outqueue>Bilkø</outqueue>
                <outqueue-pointer>1</outqueue-pointer>
        </generator>
        <generator>
                <no>2</no>
                <name>Vaskemenn-ansetter</name>
                <blueprint>Vaskemann<blueprint>
                <blueprint-pointer>2</blueprint-pointer>
                <generate>2</generate>
                <distribution>Instant</distribution>
                <distribution-pointer>3</distribution-pointer>
                <outqueue>Pauserom</outqueue>
                <outqueue-pointer>2</outqueue-pointer>
        </generator>
```

```
        </processes>

        <activities>
                <activity>
                        <no>1</no>
                        <name>Vaskemaskin</name>
                        <type>Resource</type>
                        <distribution>Vasketid</distribution>
                        <distribution-pointer>2</distribution-pointer>
                        <inqueues>
                                <queue>
                                        <name>Bilkø</name>
                                        <pointer>1</pointer>
                                </queue>
                                <queue>
                                        <name>Pauserom</name>
                                        <pointer>2</pointer>
                                </queue>
                        </inqueues>
                        <outqueues>
                                <queue>
                                        <name>Utkjørsel</name>
                                        <pointer>3</pointer>
                                </queue>
                                <queue>
                                        <name>Pauserom</name>
                                        <pointer>2</pointer>
                                </queue>
                        </outqueues>
                </activity>
        </activities>
</simulation>
</mjml>
```

D.4.2 Kø-nettverket

```
<?xml version="1.0"?>
<!DOCTYPE mjml SYSTEM "file:mjml.dtd">
<mjml>
<head>
        <name>Kønettverk</name>
        <time>10000</time>
</head>
<simulation>
        <imports>
        </imports>

        <distributions>
                <distribution>
                        <no>1</no>
                        <name>Generisk</name>
                        <type>negexp</type>
                        <mean>11</mean>
                </distribution>
                <distribution>
                        <no>2</no>
                        <name>Generisk 2</name>
                        <type>negexp</type>
                        <mean>10</mean>
                </distribution>
                <distribution>
                        <no>3</no>
                        <name>Instant</name>
                        <type>constant</type>
                        <mean>0</mean>
                </distribution>
        </distributions>

        <entities>
                <entity>
                        <no>1</no>
                        <name>Dings</name>
                        <type>dynamic</type>
                </entity>
        </entities>

        <queues>
                <queue>
                        <no>1</no>
                        <type>FifoQueue</type>
                        <name>Station 1</name>
                        <maxsize>0</maxsize>
                </queue>
                <queue>
```

```
                <no>2</no>
                <type>FifoQueue</type>
                <name>Station 2</name>
                <maxsize>0</maxsize>
        </queue>
        <queue>
                <no>3</no>
                <type>FifoQueue</type>
                <name>Station 3</name>
                <maxsize>0</maxsize>
        </queue>
        <queue>
                <no>4</no>
                <type>FifoQueue</type>
                <name>Station 4</name>
                <maxsize>0</maxsize>
        </queue>
        <queue>
                <no>5</no>
                <type>FifoQueue</type>
                <name>Station 5</name>
                <maxsize>0<maxsize>
        </queue>
        <queue>
                <no>6</no>
                <type>FifoQueue</type>
                <name>Station 6</name>
                <maxsize>0</maxsize>
        </queue>
        <queue>
                <no>7</no>
                <type>Sink</type>
                <name>Ut 1</name>
                <logfile>/home/johs/src/mjmod/quenet-sink1.csv</logfile>
        </queue>
        <queue>
                <no>8</no>
                <type>Sink</type>
                <name>Ut 2</name>
                <logfile>/home/johs/src/mjmod/quenet-sink2.csv</logfile>
        </queue>
</queues>

<processes>
        <generator>
                <no>1</no>
                <name>Inn 1</name>
                <blueprint>Dings<blueprint>
                <blueprint-pointer>1</blueprint-pointer>
                <generate>0</generate>
```

```
                <distribution>Generisk</distribution>
                <distribution-pointer>1</distribution-pointer>
                <outqueue>Station 1</outqueue>
                <outqueue-pointer>1</outqueue-pointer>
        </generator>
        <generator>
                <no>2</no>
                <name>Inn 2</name>
                <blueprint>Dings<blueprint>
                <blueprint-pointer>1</blueprint-pointer>
                <generate>0</generate>
                <distribution>Generisk</distribution>
                <distribution-pointer>1</distribution-pointer>
                <outqueue>Station 2</outqueue>
                <outqueue-pointer>2</outqueue-pointer>
        </generator>
</processes>

<activities>
        <activity>
                <no>1</no>
                <name>Splitter 1</name>
                <type>Splitter</type>
                <distribution>Generisk 2</distribution>
                <distribution-pointer>2</distribution-pointer>
                <inqueues>
                        <queue>
                                <name>Station 1</name>
                                <pointer>1</pointer>
                        </queue>
                </inqueues>
                <outqueues>
                        <queue>
                                <name>Station 3</name>
                                <pointer>3</pointer>
                        </queue>
                        <queue>
                                <name>Station 4</name>
                                <pointer>4</pointer>
                        </queue>
                </outqueues>
                <frequencies>
                        <frequency>3</frequency>
                        <frequency>1</frequency>
                </frequencies>
        </activity>
        <activity>
                <no>2</no>
                <name>Splitter 2</name>
                <type>Splitter</type>
```

```
        <distribution>Generisk 2</distribution>
        <distribution-pointer>2</distribution-pointer>
        <inqueues>
                <queue>
                        <name>Station 2</name>
                        <pointer>2</pointer>
                </queue>
        </inqueues>
        <outqueues>
                <queue>
                        <name>Station 6</name>
                        <pointer>6</pointer>
                </queue>
        </outqueues>
</activity>
<activity>
        <no>3</no>
        <name>Splitter 3</name>
        <type>Splitter</type>
        <distribution>Generisk 2</distribution>
        <distribution-pointer>2</distribution-pointer>
        <inqueues>
                <queue>
                        <name>Station 3</name>
                        <pointer>3</pointer>
                </queue>
        </inqueues>
        <outqueues>
                <queue>
                        <name>Ut 1</name>
                        <pointer>7</pointer>
                </queue>
        </outqueues>
</activity>
<activity>
        <no>4</no>
        <name>Splitter 4</name>
        <type>Splitter</type>
        <distribution>Generisk 2</distribution>
        <distribution-pointer>2</distribution-pointer>
        <inqueues>
                <queue>
                        <name>Station 4</name>
                        <pointer>4</pointer>
                </queue>
        </inqueues>
        <outqueues>
                <queue>
                        <name>Station 5</name>
                        <pointer>5</pointer>
```

```
                        </queue>
                </outqueues>
        </activity>
        <activity>
                <no>5</no>
                <name>Splitter 5</name>
                <type>Splitter</type>
                <distribution>Generisk 2</distribution>
                <distribution-pointer>2</distribution-pointer>
                <inqueues>
                        <queue>
                                <name>Station 5</name>
                                <pointer>5</pointer>
                        </queue>
                </inqueues>
                <outqueues>
                        <queue>
                                <name>Station 6</name>
                                <pointer>6</pointer>
                        </queue>
                </outqueues>
        </activity>
        <activity>
                <no>6</no>
                <name>Splitter 6</name>
                <type>Splitter</type>
                <distribution>Generisk 2</distribution>
                <distribution-pointer>2</distribution-pointer>
                <inqueues>
                        <queue>
                                <name>Station 6</name>
                                <pointer>6</pointer>
                        </queue>
                </inqueues>
                <outqueues>
                        <queue>
                                <name>Ut 2</name>
                                <pointer>8</pointer>
                        </queue>
                </outqueues>
        </activity>

    </activities>
</simulation>
</mjml>
```

Tillegg E

Utvalgt GraphXML-kode

E.1 Bilvask med én vaskemaskin

```
<?xml version="1.0"?>
<!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
<GraphXML>
        <graph isDirected="true" vendor="ModelParserGUI">
                <label>Bilvask</label>
t       <style>
        <fill tag="node" class="entity-static" xlink:href="mjml-icons/entity-static.g
        <fill tag="node" class="entity-dynamic" xlink:href="mjml-icons/entity-dynamic
        <fill tag="node" class="distribution" xlink:href="mjml-icons/distribution.gif
        <fill tag="node" class="queue" xlink:href="mjml-icons/queue.gif"/>
        <fill tag="node" class="sink" xlink:href="mjml-icons/sink.gif"/>
        <fill tag="node" class="generator" xlink:href="mjml-icons/generator.gif"/>
        <fill tag="node" class="resource" xlink:href="mjml-icons/resource.gif"/>
        <fill tag="node" class="merger" xlink:href="mjml-icons/merger.gif"/>
        <fill tag="node" class="splitter" xlink:href="mjml-icons/splitter.gif"/>
        <fill tag="node" class="tagreader" xlink:href="mjml-icons/tagreader.gif"/>
        <fill tag="node" class="tagwriter" xlink:href="mjml-icons/tagwriter.gif"/>
        <fill tag="node" class="priorityresource" xlink:href="mjml-icons/priority-res
t       </style>
        <node name="Distribution: Generisk" class="distribution">
                <size width="10" height="1"/>
                <label>Generisk(negexp with 11)</label>
        </node>
        <node name="Distribution: Vasketid" class="distribution">
                <size width="10" height="1"/>
                <label>Vasketid(constant with 10)</label>
        </node>
        <node name="Distribution: Instant" class="distribution">
                <size width="10" height="1"/>
                <label>Instant(constant with 0)</label>
        </node>
        <node name="Entity: Skitten bil" class="entity-dynamic">
```

```
        <size width="10" height="1"/>
        <label>Skitten bil(dynamic)</label>
</node>
<node name="Entity: Vaskemann" class="entity-static">
        <size width="10" height="1"/>
        <label>Vaskemann(static)</label>
</node>
<node name="Queue: Bilkø" class="queue">
        <size width="10" height="1"/>
        <label>Bilkø(FifoQueue)</label>
</node>
<node name="Queue: Pauserom" class="queue">
        <size width="10" height="1"/>
        <label>Pauserom(LifoQueue)</label>
</node>
<node name="Queue: Utkjørsel" class="sink">
        <size width="10" height="1"/>
        <label>Utkjørsel(Sink)</label>
</node>
<node name="Generator: Innkjørsel" class="generator">
        <size width="10" height="1"/>
        <label>Innkjørsel(Number to generate:Infinite)</label>
</node>
<edge source="Distribution: Generisk" target="Generator: InnkjÃ¸rsel">
        <style>
                <line linewidth="0.5" colour="orange"/>
        </style>
        <label>Get the distribution from Generisk</label>
</edge>
<edge source="Entity: Skitten bil" target="Generator: InnkjÃ¸rsel">
        <style>
                <line linewidth="0.5" colour="cyan"/>
        </style>
        <label>Get the blueprint from Generisk</label>
</edge>
<edge source="Generator: Innkjørsel" target="Queue: Bilkø">
        <style>
                <line linewidth="0.1" colour="green"/>
        </style>
        <label>Deliver the entities to Generisk</label>
</edge>
<node name="Generator: Vaskemenn-ansetter" class="generator">
        <size width="10" height="1"/>
        <label>Vaskemenn-ansetter(Number to generate:2)</label>
</node>
<edge source="Distribution: Instant" target="Generator: Vaskemenn-ansetter">
        <style>
                <line linewidth="0.5" colour="orange"/>
        </style>
        <label>Get the distribution from Instant</label>
```

```
        </edge>
        <edge source="Entity: Vaskemann" target="Generator: Vaskemenn-ansetter">
                <style>
                        <line linewidth="0.5" colour="cyan"/>
                </style>
                <label>Get the blueprint from Instant</label>
        </edge>
        <edge source="Generator: Vaskemenn-ansetter" target="Queue: Pauserom">
                <style>
                        <line linewidth="0.1" colour="green"/>
                </style>
                <label>Deliver the entities to Instant</label>
        </edge>
        <node name="Resource: Vaskemaskin" class="resource">
                <size width="10" height="1"/>
                <label>Vaskemaskin(Resource)</label>
        </node>
        <edge source="Distribution: Vasketid" target="Resource: Vaskemaskin">
                <style>
                        <line linewidth="0.1" colour="orange"/>
                </style>
                <label>Get the distribution from Vasketid</label>
        </edge>
        <edge source="Queue: Bilkø" target="Resource: Vaskemaskin">
                <style>
                        <line linewidth="0.1" colour="black"/>
                </style>
                <label>Get the entities from Bilkø</label>
        </edge>
        <edge source="Queue: Pauserom" target="Resource: Vaskemaskin">
                <style>
                        <line linewidth="0.1" colour="black"/>
                </style>
                <label>Get the entities from Pauserom</label>
        </edge>
        <edge source="Resource: Vaskemaskin" target="Queue: Utkjørsel">
                <style>
                        <line linewidth="0.1" colour="black"/>
                </style>
                <label>Release the entities into Utkjørsel</label>
        </edge>
        <edge source="Resource: Vaskemaskin" target="Queue: Pauserom">
                <style>
                        <line linewidth="0.1" colour="black"/>
                </style>
                <label>Release the entities into Pauserom</label>
        </edge>
        </graph>
</GraphXML>
```